U N I X
SYSTEM MANAGER'S MANUAL

Cover design by John Lassetter, Lucasfilm, Ltd.

UNIX SYSTEM MANAGER'S MANUAL

4.2 Berkeley Software Distribution
Virtual VAX−11 Version

March, 1984

PREFACE

This manual is part of a five volume set intended for use with the 4.2 Berkeley Software Distribution for the VAX-11 computer. While the five volumes together contain virtually the same material presented in the four volume UNIX Programmer's Manual distributed with 4.2BSD, the manuals reflect a revised organization necessitated by the large quantity of information. The documentation is divided into three logically distinct *manuals*:

- UNIX User's Manual,
- UNIX Programmer's Manual, and
- UNIX System Manager's Manual.

Each of the User and Programmer manuals are two volumes: a Reference Guide, containing relevant sections from Volume 1 of the old UNIX Programmer's Manual, and a volume of Supplementary Documents, containing pertinent material from Volume 2 of the old UNIX Programmer's Manual. The System Manager's manual consists of a single volume containing information from both Volumes 1 and 2. We acknowledge those who have assisted us in putting together these manuals. In particular, we thank Tom Ferrin for pursuing the printing particulars.

<div align="right">

M. J. Karels
S. J. Leffler

</div>

Preface to the 4.2 Berkeley distribution

This update to the 4.1 distribution of June 1981 provides support for the VAX 11/730, full networking and interprocess communication support, an entirely new file system, and many other new features. It is certainly the most ambitious release of software ever prepared here and represents many man-years of work. Bill Shannon (both at DEC and at Sun Microsystems) and Robert Elz of the University of Melbourne contributed greatly to this distribution through new device drivers and painful debugging episodes. Rob Gurwitz of BBN wrote the initial version of the code upon which the current networking support is based. Eric Allman of Britton-Lee donated countless hours to the mail system. Bill Croft (both at SRI and Sun Microsystems) aided in the debugging and development of the networking facilities. Dennis Ritchie of Bell Laboratories also contributed greatly to this distribution, providing valuable advise and guidance. Helge Skrivervik worked on the device drivers which enabled the distribution to be delivered with a TU58 console cassette and RX01 console flopppy disk, and rewrote major portions of the standalone i/o system to support formatting of non-DEC peripherals.

Numerous others contributed their time and energy in organizing the user software for release, while many groups of people on campus suffered patiently through the low spots of development. As always, we are grateful to the UNIX user community for encouragement and support.

Once again, the financial support of the Defense Advanced Research Projects Agency is gratefully acknowledged.

<div align="right">

S. J. Leffler
W. N. Joy
M. K. McKusick

</div>

UNIX System Manager's Manual

4.2 Berkeley Software Distribution, Virtual VAX−11 Version

March, 1984

This volume contains manual entries and reference documents for system administrators. The information contained in this document applys to the Virtual VAX-11 version of the system as distributed by U.C. Berkeley.

Manual References

1.   Section 8 of the UNIX† Programmer's Manual.

System Installation and Administration

2.   Installing and Operating 4.2BSD on the VAX
     The definitive reference document for those occasions when you find you need to start over again.

3.   Building 4.2BSD UNIX Systems with Config
     An in-depth discussion of the use and operation of the *config* program. This document discusses how to configure and build binary images of UNIX for your site.

4.   Disc Quotas in a UNIX Environment
     A light introduction to the care and feeding of the facilities which can be used in limiting disc resources.

5.   Fsck − The UNIX File System Check Program
     A reference document for use with the *fsck* program during times of file system distress.

6.   Sendmail Installation and Operation Guide
     The last word in installing and operating the *sendmail* program.

7.   4.2BSD Line Printer Spooler Manual
     This document describes the structure and installation procedure for the line printer spooling system.

8.   A Dial-Up Network of UNIX Systems
     Descripes UUCP, a program for communicating files between UNIX systems.

9.   UUCP Implementation
     How UUCP works, and how to administer it.

Supporting Documentation

10.   UNIX Implementation
      How the system actually works.

11.   The UNIX I/O System
      How the I/O system really works.

---

† UNIX is a trademark of Bell Laboratories.

NAME

    intro — introduction to system maintenance and operation commands

DESCRIPTION

    This section contains information related to system operation and maintenance. In particular, commands used to create new file systems, *newfs*, *mkfs*, and verify the integrity of the file systems, *fsck*, *icheck*, *dcheck*, and *ncheck* are described here. The section *format* should be consulted when formatting disk packs. The section *crash* should be consulted in understanding how to interpret system crash dumps.

LIST OF PROGRAMS

Program	Appears on Page	Description
ac	ac.8	login accounting
accton	sa.8	system accounting
adduser	adduser.8	procedure for adding new users
analyze	analyze.8	Virtual UNIX postmortem crash analyzer
arcv	arcv.8	convert archives to new format
arff	arff.8v	archiver and copier for floppy
bad144	bad144.8	read/write dec standard 144 bad sector information
badsect	badsect.8	create files to contain bad sectors
bugfiler	bugfiler.8	file bug reports in folders automatically
catman	catman.8	create the cat files for the manual
chown	chown.8	change owner
clri	clri.8	clear i-node
comsat	comsat.8c	biff server
config	config.8	build system configuration files
crash	crash.8v	what happens when the system crashes
cron	cron.8	clock daemon
dcheck	dcheck.8	file system directory consistency check
diskpart	diskpart.8	calculate default disk partition sizes
dmesg	dmesg.8	collect system diagnostic messages to form error log
drtest	drtest.8	standalone disk test program
dump	dump.8	incremental file system dump
dumpfs	dumpfs.8	dump file system information
edquota	edquota.8	edit user quotas
fastboot	fastboot.8	reboot/halt the system without checking the disks
fasthalt	fastboot.8	reboot/halt the system without checking the disks
flcopy	arff.8v	archiver and copier for floppy
format	format.8v	how to format disk packs
fsck	fsck.8	file system consistency check and interactive repair
ftpd	ftpd.8c	DARPA Internet File Transfer Protocol server
gettable	gettable.8c	get NIC format host tables from a host
getty	getty.8	set terminal mode
halt	halt.8	stop the processor
htable	htable.8	convert NIC standard format host tables
icheck	icheck.8	file system storage consistency check
ifconfig	ifconfig.8c	configure network interface parameters
implog	implog.8c	IMP log interpreter
implogd	implogd.8c	IMP logger process
init	init.8	process control initialization
kgmon	kgmon.8	generate a dump of the operating systems profile buffers
lpc	lpc.8	line printer control program
lpd	lpd.8	line printer daemon

8

makedev	makedev.8	make system special files
makekey	makekey.8	generate encryption key
mkfs	mkfs.8	construct a file system
mklost+found	mklost+found.8	make a lost+found directory for fsck
mknod	mknod.8	build special file
mkproto	mkproto.8	construct a prototype file system
mount	mount.8	mount and dismount file system
ncheck	ncheck.8	generate names from i-numbers
newfs	newfs.8	construct a new file system
pac	pac.8	printer/ploter accounting information
pstat	pstat.8	print system facts
quot	quot.8	summarize file system ownership
quotacheck	quotacheck.8	file system quota consistency checker
quotaoff	quotaon.8	turn file system quotas on and off
quotaon	quotaon.8	turn file system quotas on and off
rc	rc.8	command script for auto-reboot and daemons
rdump	rdump.8c	file system dump across the network
reboot	reboot.8	UNIX bootstrapping procedures
renice	renice.8	alter priority of running processes
repquota	repquota.8	summarize quotas for a file system
restore	restore.8	incremental file system restore
rexecd	rexecd.8c	remote execution server
rlogind	rlogind.8c	remote login server
rmt	rmt.8c	remote magtape protocol module
route	route.8c	manually manipulate the routing tables
routed	routed.8c	network routing daemon
rrestore	rrestore.8c	restore a file system dump across the network
rshd	rshd.8c	remote shell server
rwhod	rwhod.8c	system status server
rxformat	rxformat.8v	format floppy disks
sa	sa.8	system accounting
savecore	savecore.8	save a core dump of the operating system
sendmail	sendmail.8	send mail over the internet
shutdown	shutdown.8	close down the system at a given time
sticky	sticky.8	executable files with persistent text
swapon	swapon.8	specify additional device for paging and swapping
sync	sync.8	update the super block
syslog	syslog.8	log systems messages
telnetd	telnetd.8c	DARPA TELNET protocol server
tftpd	tftpd.8c	DARPA Trivial File Transfer Protocol server
trpt	trpt.8c	transliterate protocol trace
tunefs	tunefs.8	tune up an existing file system
umount	mount.8	mount and dismount file system
update	update.8	periodically update the super block
uuclean	uuclean.8c	uucp spool directory clean-up
uusnap	uusnap.8c	show snapshot of the UUCP system
vipw	vipw.8	edit the password file

8

NAME

    ac — login accounting

SYNOPSIS

    /etc/ac [—w wtmp] [—p] [—d] [people] ...

DESCRIPTION

    Ac produces a printout giving connect time for each user who has logged in during the life of the current *wtmp* file. A total is also produced. —w is used to specify an alternate *wtmp* file. —p prints individual totals; without this option, only totals are printed. —d causes a printout for each midnight to midnight period. Any *people* will limit the printout to only the specified login names. If no *wtmp* file is given, */usr/adm/wtmp* is used.

    The accounting file */usr/adm/wtmp* is maintained by *init* and *login*. Neither of these programs creates the file, so if it does not exist no connect-time accounting is done. To start accounting, it should be created with length 0. On the other hand if the file is left undisturbed it will grow without bound, so periodically any information desired should be collected and the file truncated.

FILES

    /usr/adm/wtmp

SEE ALSO

    init(8), sa(8), login(1), utmp(5).

8

NAME

 adduser — procedure for adding new users

DESCRIPTION

 A new user must choose a login name, which must not already appear in */etc/passwd*. An account can be added by editing a line into the passwd file; this must be done with the password file locked e.g. by using *vipw*(8).

 A new user is given a group and user id. User id's should be distinct across a system, since they are used to control access to files. Typically, users working on similar projects will be put in the same group. Thus at UCB we have groups for system staff, faculty, graduate students, and a few special groups for large projects. System staff is group "10" for historical reasons, and the super-user is in this group.

 A skeletal account for a new user "ernie" would look like:

     ernie::235:20:& Kovacs,508E,7925,6428202:/mnt/grad/ernie:/bin/csh

 The first field is the login name "ernie". The next field is the encrypted password which is not given and must be initialized using *passwd*(1). The next two fields are the user and group id's. Traditionally, users in group 20 are graduate students and have account names with numbers in the 200's. The next field gives information about ernie's real name, office and office phone and home phone. This information is used by the *finger*(1) program. From this information we can tell that ernie's real name is "Ernie Kovacs" (the & here serves to repeat "ernie" with appropriate capitalization), that his office is 508 Evans Hall, his extension is x2-7925, and this his home phone number is 642-8202. You can modify the *finger*(1) program if necessary to allow different information to be encoded in this field. The UCB version of finger knows several things particular to Berkeley — that phone extensions start "2—", that offices ending in "E" are in Evans Hall and that offices ending in "C" are in Cory Hall.

 The final two fields give a login directory and a login shell name. Traditionally, user files live on a file system which has the machines single letter *net*(1) address as the first of two characters. Thus on the Berkeley CS Department VAX, whose Berknet address is "csvax" abbreviated "v" the user file systems are mounted on "/va", "/vb", etc. On each such filesystem there are subdirectories there for each group of users, i.e.: "/va/staff" and "/vb/prof". This is not strictly necessary but keeps the number of files in the top level directories reasonably small.

 The login shell will default to "/bin/sh" if none is given. Most users at Berkeley choose "/bin/csh" so this is usually specified here.

 It is useful to give new users some help in getting started, supplying them with a few skeletal files such as *.profile* if they use "/bin/sh", or *.cshrc* and *.login* if they use "/bin/csh". The directory "/usr/skel" contains skeletal definitions of such files. New users should be given copies of these files which, for instance, arrange to use *tset*(1) automatically at each login.

FILES

 /etc/passwd   password file
 /usr/skel   skeletal login directory

SEE ALSO

 passwd(1), finger(1), chsh(1), chfn(1), passwd(5), vipw(8)

BUGS

 User information should be stored in its own data base separate from the password file.

NAME

analyze — Virtual UNIX postmortem crash analyzer

SYNOPSIS

/etc/analyze [−s swapfile] [−f] [−m] [−d] [−D] [−v] corefile [system]

DESCRIPTION

Analyze is the post-mortem analyzer for the state of the paging system. In order to use *analyze* you must arrange to get a image of the memory (and possibly the paging area) of the system after it crashes (see *crash*(8V)).

The *analyze* program reads the relevant system data structures from the core image file and indexing information from **/vmunix** (or the specified file) to determine the state of the paging subsystem at the point of crash. It looks at each process in the system, and the resources each is using in an attempt to determine inconsistencies in the paging system state. Normally, the output consists of a sequence of lines showing each active process, its state (whether swapped in or not), its *p0br*, and the number and location of its page table pages. Any pages which are locked while raw i/o is in progress, or which are locked because they are *intransit* are also printed. (Intransit text pages often diagnose as duplicated; you will have to weed these out by hand.)

The program checks that any pages in core which are marked as not modified are, in fact, identical to the swap space copies. It also checks for non-overlap of the swap space, and that the core map entries correspond to the page tables. The state of the free list is also checked.

Options to *analyze*:

−D   causes the diskmap for each process to be printed.

−d   causes the (sorted) paging area usage to be printed.

−f   which causes the free list to be dumped.

−m   causes the entire coremap state to be dumped.

−v   (long unused) which causes a hugely verbose output format to be used.

In general, the output from this program can be confused by processes which were forking, swapping, or exiting or happened to be in unusual states when the crash occurred. You should examine the flags fields of relevant processes in the output of a *pstat*(8) to weed out such processes.

It is possible to look at the core dump with *adb* if you do

        adb −k /vmunix /vmcore

FILES

/vmunix   default system namelist

SEE ALSO

adb(1), ps(1), crash(8V), pstat(8)

AUTHORS

Ozalp Babaoglu and William Joy

DIAGNOSTICS

Various diagnostics about overlaps in swap mappings, missing swap mappings, page table entries inconsistent with the core map, incore pages which are marked clean but differ from disk-image copies, pages which are locked or intransit, and inconsistencies in the free list.

It would be nice if this program analyzed the system in general, rather than just the paging system in particular.

8

NAME
       arcv — convert archives to new format

SYNOPSIS
       /etc/arcv file ...

DESCRIPTION
       Arcv converts archive files (see *ar*(1), *ar*(5)) from 32v and Third Berkeley editions to a new
       portable format. The conversion is done in place, and the command refuses to alter a file not
       in old archive format.

       Old archives are marked with a magic number of 0177545 at the start; new archives have a first
       line "!<arch>".

FILES
       /tmp/v*, temporary copy

SEE ALSO
       ar(1), ar(5)

NAME
     arff, flcopy — archiver and copier for floppy

SYNOPSIS
     /etc/arff [key] [name ...]
     /etc/flcopy [−h] [−t*n*]

DESCRIPTION
     Arff saves and restores files on the console floppy disk. Its actions are controlled by the *key* argument. The *key* is a string of characters containing at most one function letter and possibly one or more function modifiers. Other arguments to the command are file names specifying which files are to be dumped or restored.

     Files names have restrictions, because of radix50 considerations. They must be in the form 1-6 alphanumerics followed by "." followed by 0-3 alphanumerics. Case distinctions are lost. Only the trailing component of a pathname is used.

     The function portion of the key is specified by one of the following letters:

     r   The named files are replaced where found on the floppy, or added taking up the minimal possible portion of the first empty spot on the floppy.

     x   The named files are extracted from the floppy.

     d   The named files are deleted from the floppy. Arff will combine contiguous deleted files into one empty entry in the rt-11 directory.

     t   The names of the specified files are listed each time they occur on the floppy. If no file argument is given, all of the names on the floppy are listed.

     The following characters may be used in addition to the letter which selects the function desired.

     v   The v (verbose) option, when used with the t function gives more information about the floppy entries than just the name.

     f   causes *arff* to use the next argument as the name of the archive instead of /dev/floppy.

     m   causes *arff* not to use the mapping algorithm employed in interleaving sectors around a floppy disk. In conjunction with the f option it may be used for extracting files from rt11 formatted cartridge disks, for example. It may also be used to speed up reading from and writing to rx02 floppy disks, by using the 'c' device instead of the 'b' device.

     c   causes *arff* to create a new directory on the floppy, effectively deleting all previously existing files.

     Flcopy copies the console floppy disk (opened as '/dev/floppy') to a file created in the current directory, named "floppy", then prints the message "Change Floppy, hit return when done". Then *flcopy* copies the local file back out to the floppy disk.

     The −h option to *flcopy* causes it to open a file named "floppy" in the current directory and copy it to */dev/floppy;* the −t option causes only the first *n* tracks to participate in a copy.

     Arff may also be used with the console TU58 cassettes on the 11/730. To do so, the m key must be specified. Normally, the f key is also used.

FILES
     /dev/floppy or /dev/rrx??
     floppy (in current directory)

8

SEE ALSO
       fl(4), rx(4), rxformat(8V)

AUTHORS
       Keith Sklower, Richard Tuck

BUGS
       Floppy errors are handled ungracefully; *Arff* does not handle multi-segment rt11 directories.

NAME

bad144 — read/write dec standard 144 bad sector information

SYNOPSIS

/etc/bad144 [−f] disktype disk [sno [bad ...]]

DESCRIPTION

Bad144 can be used to inspect the information stored on a disk that is used by the disk drivers to implement bad sector forwarding. The format of the information is specified by DEC standard 144, as follows.

The bad sector information is located in the first 5 even numbered sectors of the last track of the disk pack. There are five identical copies of the information, described by the *dkbad* structure.

Replacement sectors are allocated starting with the first sector before the bad sector information and working backwards towards the beginning of the disk. A maximum of 126 bad sectors are supported. The position of the bad sector in the bad sector table determines which replacement sector it corresponds to.

The bad sector information and replacement sectors are conventionally only accessible through the "c" file system partition of the disk. If that partition is used for a file system, the user is responsible for making sure that it does not overlap the bad sector information or any replacement sectors.

The bad sector structure is as follows:

```
struct dkbad {
        long      bt_csn;              /* cartridge serial number */
        u_short   bt_mbz;              /* unused; should be 0 */
        u_short   bt_flag;            /* -1 => alignment cartridge */
        struct bt_bad {
                u_short bt_cyl;       /* cylinder number of bad sector */
                u_short bt_trksec;    /* track and sector number */
        } bt_bad[126];
};
```

Unused slots in the *bt_bad* array are filled with all bits set, a putatively illegal value.

Bad144 is invoked by giving a device type (e.g. rk07, rm03, rm05, etc.), and a device name (e.g. hk0, hp1, etc.). It reads the first sector of the last track of the corresponding disk and prints out the bad sector information. It may also be invoked giving a serial number for the pack and a list of bad sectors, and will then write the supplied information onto the same location. Note, however, that *bad144* does not arrange for the specified sectors to be marked bad in this case. This option should only be used to restore known bad sector information which was destroyed.

If the disk is an RP06, Fujitsu Eagle, or Ampex Capricorn on a Massbus, the −f option may be used to mark the bad sectors as "bad". NOTE: **this can only be done safely when there is no other disk activity,** preferably while running single-user. Otherwise, new bad sectors can be added only by running a formatter. Note that the order in which the sectors are listed determines which sectors used for replacements; if new sectors are being added to the list on a drive that is in use, care should be taken that replacements for existing bad sectors have the correct contents.

SEE ALSO

badsect(8), format(8V)

8

BUGS

It should be possible to format disks on-line under UNIX.

It should be possible to mark bad sectors on drives of all type.

On an 11/750, the standard bootstrap drivers used to boot the system do not understand bad sectors, handle ECC errors, or the special SSE (skip sector) errors of RM80 type disks. This means that none of these errors can occur when reading the file /vmunix to boot. Sectors 0-15 of the disk drive must also not have any of these errors.

The drivers which write a system core image on disk after a crash do not handle errors; thus the crash dump area must be free of errors and bad sectors.

NAME
     badsect — create files to contain bad sectors

SYNOPSIS
     /etc/badsect bbdir sector ...

DESCRIPTION
     Badsect makes a file to contain a bad sector. Normally, bad sectors are made inaccessible by the standard formatter, which provides a forwarding table for bad sectors to the driver; see *bad144*(8) for details. If a driver supports the bad blocking standard it is much preferable to use that method to isolate bad blocks, since the bad block forwarding makes the pack appear perfect, and such packs can then be copied with *dd*(1). The technique used by this program is also less general than bad block forwarding, as *badsect* can't make amends for bad blocks in the i-list of file systems or in swap areas.

     On some disks, adding a sector which is suddenly bad to the bad sector table currently requires the running of the standard DEC formatter. Thus to deal with a newly bad block or on disks where the drivers do not support the bad-blocking standard *badsect* may be used to good effect.

     Badsect is used on a quiet file system in the following way: First mount the file system, and change to its root directory. Make a directory BAD there. Run *badsect* giving as argument the BAD directory followed by all the bad sectors you wish to add. (The sector numbers must be relative to the beginning of the file system, but this is not hard as the system reports relative sector numbers in its console error messages.) Then change back to the root directory, unmount the file system and run *fsck*(8) on the file system. The bad sectors should show up in two files or in the bad sector files and the free list. Have *fsck* remove files containing the offending bad sectors, but do not have it remove the BAD/*nnnnn* files. This will leave the bad sectors in only the BAD files.

     Badsect works by giving the specified sector numbers in a *mknod*(2) system call, creating an illegal file whose first block address is the block containing bad sector and whose name is the bad sector number. When it is discovered by *fsck* it will ask "HOLD BAD BLOCK"? A positive response will cause *fsck* to convert the inode to a regular file containing the bad block.

SEE ALSO
     bad144(8), fsck(8), format(8V)

DIAGNOSTICS
     Badsect refuses to attach a block that resides in a critical area or is out of range of the file system. A warning is issued if the block is already in use.

BUGS
     If more than one sector which comprise a file system fragment are bad, you should specify only one of them to *badsect,* as the blocks in the bad sector files actually cover all the sectors in a file system fragment.

8

NAME
       bugfiler — file bug reports in folders automatically

SYNOPSIS
       bugfiler [mail directory]

DESCRIPTION
       Bugfiler is a program to automatically intercept bug reports, summarize them and store them in
       the appropriate sub directories of the mail directory specified on the command line or the (sys-
       tem dependent) default. It is designed to be compatible with the Rand MH mail system.
       Bugfiler is normally invoked by the mail delivery program through *aliases*(5) with a line such as
       the following in /usr/lib/aliases.

              bugs:"bugfiler /usr/bugs/mail"

       It reads the message from the standard input or the named file, checks the format and returns
       mail acknowledging receipt or a message indicating the proper format. Valid reports are then
       summarized and filed in the appropriate folder. Users can then log onto the system and check
       the summary file for bugs that pertain to them. Bug reports are submitted in RFC822 format
       and must contain the following header lines:

              Date: <date the report is received>
              From: <valid return address>
              Subject: <short summary of the problem>
              Index: <source directory>/<source file> <version> [Fix]

       In addition, the body of the message must contain a line which begins with "Description:" fol-
       lowed by zero or more lines describing the problem in detail and a line beginning with
       "Repeat-By:" followed by zero or more lines describing how to repeat the problem. If the key-
       word 'Fix' is specified in the 'Index' line, then there must also be a line beginning with "Fix:"
       followed by a diff of the old and new source files or a description of what was done to fix the
       problem.

       The 'Index' line is the key to the filing mechanism. The source directory name must match one
       of the folder names in the mail directory. The message is then filed in this folder and a line
       appended to the summary file in the following format:

              <folder name>/<message number>   <Index info>
                                               <Subject info>

FILES
       /usr/new/lib/mh/deliver   mail delivery program
       /usr/new/lib/mh/unixtomh   converts unix mail format to mh format
       maildir/.ack   the message sent in acknowledgement
       maildir/.format   the message sent when format errors are detected
       maildir/summary   the summary file
       maildir/Bf??????   temporary copy of the input message
       maildir/Rp??????   temporary file for the reply message.

SEE ALSO
       mh(1), newaliases(1), aliases(5)

BUGS
       Since mail can be forwarded in a number of different ways, *bugfiler* does not recognize for-
       warded mail and will reply/complain to the forwarder instead of the original sender unless there
       is a 'Reply-To' field in the header.

       Duplicate messages should be discarded or recognized and put somewhere else.

NAME

catman — create the cat files for the manual

SYNOPSIS

/etc/catman [−p] [−n] [−w] [sections]

DESCRIPTION

Catman creates the preformatted versions of the on-line manual from the nroff input files. Each manual page is examined and those whose preformatted versions are missing or out of date are recreated. If any changes are made, *catman* will recreate the **/usr/lib/whatis** database.

If there is one parameter not starting with a '−', it is take to be a list of manual sections to look in. For example

catman 123

will cause the updating to only happen to manual sections 1, 2, and 3.

Options:

−n   prevents creations of **/usr/lib/whatis**.

−p   prints what would be done instead of doing it.

−w   causes only the **/usr/lib/whatis** database to be created. No manual reformatting is done.

FILES

/usr/man/man?/*.*	raw (nroff input) manual sections
/usr/man/cat?/*.*	preformatted manual pages
/usr/lib/makewhatis	commands to make whatis database

SEE ALSO

man(1)

BUGS

Acts oddly on nights with full moons.

8

NAME
     chown — change owner

SYNOPSIS
     /etc/chown [−f] owner file ...

DESCRIPTION
     Chown changes the owner of the *files* to *owner*. The owner may be either a decimal UID or a
     login name found in the password file.

     Only the super-user can change owner, in order to simplify accounting procedures. No errors
     are reported when the −f (force) option is given.

FILES
     /etc/passwd

SEE ALSO
     chgrp(1), chown(2), passwd(5), group(5)

NAME

    clri — clear i-node

SYNOPSIS

    /etc/clri filesystem i-number ...

DESCRIPTION

    N.B.: *Clri* is obsoleted for normal file system repair work by *fsck*(8).

    Clri writes zeros on the i-nodes with the decimal *i-numbers* on the *filesystem*. After *clri*, any blocks in the affected file will show up as 'missing' in an *icheck*(8) of the *filesystem*.

    Read and write permission is required on the specified file system device. The i-node becomes allocatable.

    The primary purpose of this routine is to remove a file which for some reason appears in no directory. If it is used to zap an i-node which does appear in a directory, care should be taken to track down the entry and remove it. Otherwise, when the i-node is reallocated to some new file, the old entry will still point to that file. At that point removing the old entry will destroy the new file. The new entry will again point to an unallocated i-node, so the whole cycle is likely to be repeated again and again.

SEE ALSO

    icheck(8)

BUGS

    If the file is open, *clri* is likely to be ineffective.

8

NAME
comsat — biff server

SYNOPSIS
/etc/comsat

DESCRIPTION
Comsat is the server process which listens for reports of incoming mail and notifies users if they have requested this service. *Comsat* listens on a datagram port associated with the "biff" service specification (see *services*(5)) for one line messages of the form

user@mailbox-offset

If the *user* specified is logged in to the system and the associated terminal has the owner execute bit turned on (by a "biff y"), the *offset* is used as a seek offset into the appropriate mailbox file and the first 7 lines or 560 characters of the message are printed on the user's terminal. Lines which appear to be part of the message header other than the "From", "To", "Date", or "Subject" lines are not included in the displayed message.

FILES
/etc/utmp   to find out who's logged on and on what terminals

SEE ALSO
biff(1)

BUGS
The message header filtering is prone to error.

Users should be notified of mail which arrives on other machines than the one they are currently logged in to.

The notification should appear in a separate window so it does not mess up the screen.

NAME

config — build system configuration files

SYNOPSIS

/etc/config [—p] *config_file*

DESCRIPTION

Config builds a set of system configuration files from a short file which describes the sort of system that is being configured. It also takes as input a file which tells *config* what files are needed to generate a system. This can be augmented by a configuration specific set of files that give alternate files for a specific machine. (see the FILES section below) If the —p option is supplied, *config* will configure a system for profiling; c.f. *kgmon*(8), *gprof*(1).

Config should be run from the **conf** subdirectory of the system source (usually /sys/conf). *Config* assumes that there is already a directory *../config_file* created and it places all its output files in there. The output of *config* consists of a number files: **ioconf.c** contains a description of what i/o devices are attached to the system,; **ubglue.s** contains a set of interrupt service routines for devices attached to the UNIBUS; **makefile** is a file used by *make*(1) in building the system; a set of header files which contain the number of various devices that will be compiled into the system; and a set of swap configuration files which contain definitions for the disk areas to be used for swapping, the root file system, argument processing, and system dumps.

After running *config*, it is necessary to run "make depend" in the directory where the new makefile was created. *Config* reminds you of this when it completes.

If you get any other error messages from *config,* you should fix the problems in your configuration file and try again. If you try to compile a system that had configuration errors, you will likely meet with failure.

FILES

/sys/conf/makefile.vax generic makefile for the VAX
/sys/conf/files list of common files system is built from
/sys/conf/files.vax   list of VAX specific files
/sys/conf/devices.vax   name to major device mapping file for the VAX
/sys/conf/files.ERNIE list of files specific to ERNIE system

SEE ALSO

"Building 4.2BSD UNIX System with Config"
The SYNOPSIS portion of each device in section 4.

BUGS

The line numbers reported in error messages are usually off by one.

8

NAME

　　crash — what happens when the system crashes

DESCRIPTION

　　This section explains what happens when the system crashes and how you can analyze crash dumps.

　　When the system crashes voluntarily it prints a message of the form

　　　　panic: why i gave up the ghost

　　on the console, takes a dump on a mass storage peripheral, and then invokes an automatic reboot procedure as described in *reboot*(8). (If auto-reboot is disabled on the front panel of the machine the system will simply halt at this point.) Unless some unexpected inconsistency is encountered in the state of the file systems due to hardware or software failure the system will then resume multi-user operations.

　　The system has a large number of internal consistency checks; if one of these fails, then it will panic with a very short message indicating which one failed.

　　The most common cause of system failures is hardware failure, which can reflect itself in different ways. Here are the messages which you are likely to encounter, with some hints as to causes. Left unstated in all cases is the possibility that hardware or software error produced the message in some unexpected way.

IO err in push
hard IO err in swap

　　The system encountered an error trying to write to the paging device or an error in reading critical information from a disk drive. You should fix your disk if it is broken or unreliable.

timeout table overflow

　　This really shouldn't be a panic, but until we fix up the data structure involved, running out of entries causes a crash. If this happens, you should make the timeout table bigger.

KSP not valid
SBI fault
CHM? in kernel

　　These indicate either a serious bug in the system or, more often, a glitch or failing hardware. If SBI faults recur, check out the hardware or call field service. If the other faults recur, there is likely a bug somewhere in the system, although these can be caused by a flakey processor. Run processor microdiagnostics.

machine check %x:

　　description

　　machine dependent machine-check information

　　We should describe machine checks, and will someday. For now, ask someone who knows (like your friendly field service people).

trap type %d, code=%d, pc=%x

　　A unexpected trap has occurred within the system; the trap types are:

0	reserved addressing fault
1	privileged instruction fault
2	reserved operand fault
3	bpt instruction fault
4	xfc instruction fault
5	system call trap

6	arithmetic trap
7	ast delivery trap
8	segmentation fault
9	protection fault
10	trace trap
11	compatibility mode fault
12	page fault
13	page table fault

The favorite trap types in system crashes are trap types 8 and 9, indicating a wild refer-
ence. The code is the referenced address, and the pc at the time of the fault is printed.
These problems tend to be easy to track down if they are kernel bugs since the proces-
sor stops cold, but random flakiness seems to cause this sometimes.

init died

The system initialization process has exited. This is bad news, as no new users will
then be able to log in. Rebooting is the only fix, so the system just does it right away.

That completes the list of panic types you are likely to see.

When the system crashes it writes (or at least attempts to write) an image of memory into the
back end of the primary swap area. After the system is rebooted, the program *savecore*(8) runs
and preserves a copy of this core image and the current system in a specified directory for later
perusal. See *savecore*(8) for details.

To analyze a dump you should begin by running *adb*(1) with the −k flag on the core dump.
Normally the command "*(intstack-4)$c" will provide a stack trace from the point of the crash
and this will provide a clue as to what went wrong. A more complete discussion of system
debugging is impossible here. See, however, "Using ADB to Debug the UNIX Kernel".

SEE ALSO

adb(1), analyze(8), reboot(8)
VAX 11/780 System Maintenance Guide for more information about machine checks.
Using ADB to Debug the UNIX Kernel

8

NAME
cron — clock daemon

SYNOPSIS
/etc/cron

DESCRIPTION
Cron executes commands at specified dates and times according to the instructions in the file /usr/lib/crontab. Since *cron* never exits, it should only be executed once. This is best done by running *cron* from the initialization process through the file /etc/rc; see *init*(8).

Crontab consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns to specify the minute (0-59), hour (0-23), day of the month (1-31), month of the year (1-12), and day of the week (1-7 with 1=Monday). Each of these patterns may contain a number in the range above; two numbers separated by a minus meaning a range inclusive; a list of numbers separated by commas meaning any of the numbers; or an asterisk meaning all legal values. The sixth field is a string that is executed by the Shell at the specified times. A percent character in this field is translated to a new-line character. Only the first line (up to a % or end of line) of the command field is executed by the Shell. The other lines are made available to the command as standard input.

Crontab is examined by *cron* every minute.

FILES
/usr/lib/crontab

8

NAME

    dcheck — file system directory consistency check

SYNOPSIS

    /etc/dcheck [−l numbers] [filesystem]

DESCRIPTION

    N.B.: *Dcheck* is obsoleted for normal consistency checking by *fsck*(8).

    Dcheck reads the directories in a file system and compares the link-count in each i-node with the number of directory entries by which it is referenced. If the file system is not specified, a set of default file systems is checked.

    The −l flag is followed by a list of i-numbers; when one of those i-numbers turns up in a directory, the number, the i-number of the directory, and the name of the entry are reported.

    The program is fastest if the raw version of the special file is used, since the i-list is read in large chunks.

FILES

    Default file systems vary with installation.

SEE ALSO

    fsck(8), icheck(8), fs(5), clri(8), ncheck(8)

DIAGNOSTICS

    When a file turns up for which the link-count and the number of directory entries disagree, the relevant facts are reported. Allocated files which have 0 link-count and no entries are also listed. The only dangerous situation occurs when there are more entries than links; if entries are removed, so the link-count drops to 0, the remaining entries point to thin air. They should be removed. When there are more links than entries, or there is an allocated file with neither links nor entries, some disk space may be lost but the situation will not degenerate.

BUGS

    Since *dcheck* is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.

    Dcheck is obsoleted by *fsck* and remains for historical reasons.

8

NAME

diskpart — calculate default disk partition sizes

SYNOPSIS

/etc/diskpart [−p] [−d] disk-type

DESCRIPTION

Diskpart is used to calculate the disk partition sizes based on the default rules used at Berkeley. If the −p option is supplied, tables suitable for inclusion in a device driver are produced. If the −d option is supplied, an entry suitable for inclusion in the disk description file *letcldisktab* is generated; c.f. *disktab*(5). Space is always left in the last partition on the disk for a bad sector forwarding table. The space reserved is one track for the replicated copies of the table and sufficient tracks to hold a pool of 126 sectors to which bad sectors are mapped. For more information, see *bad144*(8).

The disk partition sizes are based on the total amount of space on the disk as give in the table below (all values are supplied in units of 512 byte sectors). The 'c' partition is, by convention, used to access the entire physical disk, including the space reserved for the bad sector forwarding table. In normal operation, either the 'g' partition is used, or the 'd', 'e', and 'f' partitions are used. The 'g' and 'f' partitions are variable sized, occupying whatever space remains after allocation of the fixed sized partitions. If the disk is smaller than 20 Megabytes, then *diskpart* aborts with the message "disk too small, calculate by hand".

Partition	20-60 MB	61-205 MB	206-355 MB	356+ MB
a	15884	15884	15884	15884
b	10032	33440	33440	66880
d	15884	15884	15884	15884
e	unused	55936	55936	307200
h	unused	unused	291346	291346

If an unknown disk type is specified, *diskpart* will prompt for the required disk geometry information.

SEE ALSO

disktab(5), bad144(8)

BUGS

Certain default partition sizes are based on historical artifacts (e.g. RP06), and may result in unsatisfactory layouts.

When using the −d flag, alternate disk names are not included in the output.

Does not understand how to handle drives attached to a UDA50.

8

NAME

    dmesg — collect system diagnostic messages to form error log

SYNOPSIS

    /etc/dmesg [—]

DESCRIPTION

    Dmesg looks in a system buffer for recently printed diagnostic messages and prints them on the standard output. The messages are those printed by the system when device (hardware) errors occur and (occasionally) when system tables overflow non-fatally. If the — flag is given, then *dmesg* computes (incrementally) the new messages since the last time it was run and places these on the standard output. This is typically used with *cron*(8) to produce the error log */usr/adm/messages* by running the command

        /etc/dmesg — >> /usr/adm/messages

    every 10 minutes.

FILES

    /usr/adm/messages   error log (conventional location)
    /usr/adm/msgbuf   scratch file for memory of — option

BUGS

    The system error message buffer is of small finite size. As *dmesg* is run only every few minutes, not all error messages are guaranteed to be logged. This can be construed as a blessing rather than a curse.

    Error diagnostics generated immediately before a system crash will never get logged.

NAME

drtest — standalone disk test program

DESCRIPTION

Drtest is a standalone program used to read a disk track by track. It was primarily intended as a test program for new standalone drivers, but has shown useful in other contexts as well, such as verifying disks and running speed tests. For example, when a disk has been formatted (by format(8)), you can check that hard errors has been taken care of by running *drtest*. No hard errors should be found, but in many cases quite a few soft ECC errors will be reported.

While *drtest* is running, the cylinder number is printed on the console for every 10th cylinder read.

EXAMPLE

A sample run of *drtest* is shown below. In this example (using a 750), *drtest* is loaded from the root file system; usually it will be loaded from the machine's console storage device. Boldface means user input. As usual, "#" and "@" may be used to edit input.

```
>>>B/3
%%
loading hk(0,0)boot
Boot
: hk(0,0)drtest
Test program for stand-alone up and hp driver

Debugging level (1=bse, 2=ecc, 3=bse+ecc)?
Enter disk name [type(adapter,unit), e.g. hp(1,3)]? hp(0,0)
Device data: #cylinders=1024, #tracks=16, #sectors=32
Testing hp(0,0), chunk size is 16384 bytes.
(chunk size is the number of bytes read per disk access)
Start ...Make sure hp(0,0) is online
  ...
(errors are reported as they occur)

  ...
(...program restarts to allow checking other disks)
(...to abort halt machine with ^P)
```

DIAGNOSTICS

The diagnostics are intended to be self explanatory. Note, however, that the device number in the diagnostic messages is identified as *typeX* instead of *type(a,u)* where $X = a \cdot 8 + u$, e.g., hp(1,3) becomes hp11.

SEE ALSO

format(8), bad144(8)

AUTHOR

Helge Skrivervik

NAME

dump — incremental file system dump

SYNOPSIS

/etc/dump [key [*argument ...*] filesystem]

DESCRIPTION

Dump copies to magnetic tape all files changed after a certain date in the *filesystem*. The *key* specifies the date and other options about the dump. *Key* consists of characters from the set **0123456789fusdWn.**

0—9 This number is the 'dump level'. All files modified since the last date stored in the file */etc/dumpdates* for the same filesystem at lesser levels will be dumped. If no date is determined by the level, the beginning of time is assumed; thus the option 0 causes the entire filesystem to be dumped.

f   Place the dump on the next *argument* file instead of the tape. If the name of the file is "—", *dump* writes to standard output.

u   If the dump completes successfully, write the date of the beginning of the dump on file */etc/dumpdates*. This file records a separate date for each filesystem and each dump level. The format of */etc/dumpdates* is readable by people, consisting of one free format record per line: filesystem name, increment level and *ctime(3)* format dump date. */etc/dumpdates* may be edited to change any of the fields, if necessary.

s   The size of the dump tape is specified in feet. The number of feet is taken from the next *argument*. When the specified size is reached, *dump* will wait for reels to be changed. The default tape size is 2300 feet.

d   The density of the tape, expressed in BPI, is taken from the next *argument*. This is used in calculating the amount of tape used per reel. The default is 1600.

W   *Dump* tells the operator what file systems need to be dumped. This information is gleaned from the files */etc/dumpdates* and */etc/fstab*. The W option causes *dump* to print out, for each file system in */etc/dumpdates* the most recent dump date and level, and highlights those file systems that should be dumped. If the W option is set, all other options are ignored, and *dump* exits immediately.

w   Is like W, but prints only those filesystems which need to be dumped.

n   Whenever *dump* requires operator attention, notify by means similar to a *wall*(1) all of the operators in the group "operator".

If no arguments are given, the *key* is assumed to be 9u and a default file system is dumped to the default tape.

Dump requires operator intervention on these conditions: end of tape, end of dump, tape write error, tape open error or disk read error (if there are more than a threshold of 32). In addition to alerting all operators implied by the n key, *dump* interacts with the operator on *dump's* control terminal at times when *dump* can no longer proceed, or if something is grossly wrong. All questions *dump* poses must be answered by typing "yes" or "no", appropriately.

Since making a dump involves a lot of time and effort for full dumps, *dump* checkpoints itself at the start of each tape volume. If writing that volume fails for some reason, *dump* will, with operator permission, restart itself from the checkpoint after the old tape has been rewound and removed, and a new tape has been mounted.

Dump tells the operator what is going on at periodic intervals, including usually low estimates of the number of blocks to write, the number of tapes it will take, the time to completion, and the time to the tape change. The output is verbose, so that others know that the terminal controlling *dump* is busy, and will be for some time.

8

Now a short suggestion on how to perform dumps. Start with a full level 0 dump

    dump 0un

Next, dumps of active file systems are taken on a daily basis, using a modified Tower of Hanoi algorithm, with this sequence of dump levels:

        3 2 5 4 7 6 9 8 9 9 ...

For the daily dumps, a set of 10 tapes per dumped file system is used on a cyclical basis. Each week, a level 1 dump is taken, and the daily Hanoi sequence repeats with 3. For weekly dumps, a set of 5 tapes per dumped file system is used, also on a cyclical basis. Each month, a level 0 dump is taken on a set of fresh tapes that is saved forever.

FILES

/dev/rrp1g	default filesystem to dump from
/dev/rmt8	default tape unit to dump to
/etc/ddate	old format dump date record (obsolete after −J option)
/etc/dumpdates	new format dump date record
/etc/fstab	dump table: file systems and frequency
/etc/group	to find group *operator*

SEE ALSO

restore(8), dump(5), fstab(5)

DIAGNOSTICS

Many, and verbose.

BUGS

Sizes are based on 1600 BPI blocked tape; the raw magtape device has to be used to approach these densities. Fewer than 32 read errors on the filesystem are ignored. Each reel requires a new process, so parent processes for reels already written just hang around until the entire tape is written.

It would be nice if *dump* knew about the dump sequence, kept track of the tapes scribbled on, told the operator which tape to mount when, and provided more assistance for the operator running *restore*.

8

NAME

    dumpfs − dump file system information

SYNOPSIS

    dumpfs *filesys|device*

DESCRIPTION

    Dumpfs prints out the super block and cylinder group information for the file system or special device specified. The listing is very long and detailed. This command is useful mostly for finding out certain file system information such as the file system block size and minimum free space percentage.

SEE ALSO

    fs(5), disktab(5), tunefs(8), newfs(8), fsck(8)

8

NAME

      edquota — edit user quotas

SYNOPSIS

      edquota [−p *proto-user*] *users...*

DESCRIPTION

      Edquota is a quota editor. One or more users may be specified on the command line. For each
      user a temporary file is created with an ASCII representation of the current disc quotas for that
      user and an editor is then invoked on the file. The quotas may then be modified, new quotas
      added, etc. Upon leaving the editor, *edquota* reads the temporary file and modifies the binary
      quota files to reflect the changes made.

      If the −p option is specified, *edquota* will duplicate the quotas of the prototypical user specified
      for each user specified. This is the normal mechanism used to initialize quotas for groups of
      users.

      The editor invoked is *vi*(1) unless the environment variable EDITOR specifies otherwise.

      Only the super-user may edit quotas.

FILES

      quotas   at the root of each file system with quotas
      /etc/fstab   to find file system names and locations

SEE ALSO

      quota(1), quota(2), quotacheck(8), quotaon(8), repquota(8)

DIAGNOSTICS

      Various messages about inaccessible files; self-explanatory.

BUGS

      The format of the temporary file is inscrutable.

NAME

　　　fastboot, fasthalt — reboot/halt the system without checking the disks

SYNOPSIS

　　　/etc/fastboot [*boot-options*]
　　　/etc/fasthalt [*halt-options*]

DESCRIPTION

　　　Fastboot and *fasthalt* are shell scripts which reboot and halt the system without checking the file
　　　systems. This is done by creating a file */fastboot*, then invoking the *reboot* program. The sys-
　　　tem startup script, */etc/rc*, looks for this file and, if present, skips the normal invocation of
　　　fsck(8).

SEE ALSO

　　　halt(8), reboot(8), rc(8)

8

NAME

format — how to format disk packs

DESCRIPTION

There are two ways to format disk packs. The simplest is to use the *format* program. The alternative is to use the DEC standard formatting software which operates under the DEC diagnostic supervisor. This manual page describes the operation of *format*, then concludes with some remarks about using the DEC formatter.

Format is a standalone program used to format and check disks prior to constructing file systems. In addition to the formatting operation, *format* records any bad sectors encountered according to DEC standard 144. Formatting is performed one track at a time by writing the appropriate headers and a test pattern and then checking the sector by reading and verifying the pattern, using the controller's ECC for error detection. A sector is marked bad if an unrecoverable media error is detected, or if a correctable ECC error greater than 5 bits in length is detected (such errors are indicated as "ECC" in the summary printed upon completing the format operation). After the entire disk has been formatted and checked, the total number of errors are reported, any bad sectors and skip sectors are marked, and a bad sector forwarding table is written to the disk in the first five even numbered sectors of the last track. *Format* may be used on any UNIBUS or MASSBUS drive supported by the *up* and *hp* device drivers which uses 4-byte headers (everything except RP's).

The test pattern used during the media check may be selected from one of: 0xf00f (RH750 worst case), 0xec6d (media worst case), and 0xa5a5 (alternating 1's and 0's). Normally the media worst case pattern is used.

Format also has an option to perform an extended "severe burnin," which makes 46 passes using different patterns. Using this option, sectors with any errors of any size are marked bad. This test runs for many hours, depending on the disk and processor.

Each time *format* is run a completely new bad sector table is generated based on errors encountered while formatting. The device driver, however, will always attempt to read any existing bad sector table when the device is first opened. Thus, if a disk pack has never previously been formatted, or has been formatted with different sectoring, five error messages will be printed when the driver attempts to read the bad sector table; these diagnostics should be ignored.

Formatting a 400 megabyte disk on a MASSBUS disk controller usually takes about 20 minutes. Formatting on a UNIBUS disk controller takes significantly longer. For every hundredth cylinder formatted *format* prints a message indicating the current cylinder being formatted. (This message is just to reassure people that nothing is is amiss.)

Format uses the standard notation of the standalone i/o library in identifying a drive to be formatted. A drive is specified as $zz(x,y)$, where zz refers to the controller type (either *hp* or *up*), x is the unit number of the drive; 8 times the UNIBUS or MASSBUS adaptor number plus the MASSBUS drive number or UNIBUS drive unit number; and y is the file system partition on drive x (this should always be 0). For example, "hp(1,0)" indicates that drive 1 on MASSBUS adaptor 0 should be formatted; while "up(10,0)" indicates UNIBUS drive 2 on UNIBUS adaptor 1 should be formatted.

Before each formatting attempt, *format* prompts the user in case debugging should be enabled in the appropriate device driver. A carriage return disables debugging information.

Format should be used prior to building file systems (with *newfs*(8)) to insure all sectors with uncorrectable media errors are remapped. If a drive develops uncorrectable defects after formatting, the program *badsect*(8) must be used.

EXAMPLE

A sample run of *format* is shown below. In this example (using a VAX-11/780), *format* is loaded from the console floppy; on an 11/750 *format* will be loaded from the root file system.

Boldface means user input. As usual, "#" and "@" may be used to edit input.

```
>>>L FORMAT
              LOAD DONE, 00004400 BYTES LOADED
>>>S 2
Disk format/check utility
```

Enable debugging (0=none, 1=bse, 2=ecc, 3=bse+ecc)? **0**
Device to format? **hp(8,0)**
(*error messages may occur as old bad sector table is read*)
Formatting drive hp0 on adaptor 1: verify (yes/no)? yes
Device data: #cylinders=842, #tracks=20, #sectors=48
Available test patterns are:
                1 - (f00f) rh750 worst case
                2 - (ec6d) media worst case
                3 - (a5a5) alternating 1's and 0's
                4 - (ffff) Severe burnin (takes several hours)
Pattern (one of the above, other to restart)? **2**
Start formatting...make sure the drive is online

   ...
(*soft ecc's and other errors are reported as they occur*)

   ...
(*if 4 write check errors were found, the program terminates like this...*)

   ...
Errors:
Write check: 4
Bad sector: 0
ECC: 0
Skip sector: 0
Total of 4 hard errors found.
Writing bad sector table at block 808271
(*808271 is the block # of the first block in the bad sector table*)
Done
(*...program restarts to allow formatting other disks*)
(*...to abort halt machine with ^P*)

DIAGNOSTICS

The diagnostics are intended to be self explanatory.

USING DEC SOFTWARE TO FORMAT

Warning: These instructions are for people with 11/780 CPU's. The steps needed for 11/750 or 11/730 cpu's are similar, but not covered in detail here.

The formatting procedures are different for each type of disk. Listed here are the formatting procedures for RK07's, RP0X, and RM0X disks.

You should shut down UNIX and halt the machine to do any disk formatting. Make certain you put in the pack you want formatted. It is also a good idea to spin down or write protect the disks you don't want to format, just in case.

Formatting an RK07. Load the console floppy labeled, "RX11 VAX DSK LD DEV #1" in the console disk drive, and type the following commands:
```
>>>BOOT
DIAGNOSTIC SUPERVISOR.  ZZ-ESSAA-X5.0-119  23-JAN-1980 12:44:40.03
DS>ATTACH DW780 SBI DW0 3 5
```

8

        DS>ATTACH RK611 DMA
        DS>ATTACH RK07 DW0 DMA0
        DS>SELECT DMA0
        DS>LOAD EVRAC
        DS>START/SEC:PACKINIT

Formatting an RP0X. Follow the above procedures except that the ATTACH and SELECT lines should read:

        DS>ATTACH RH780 SBI RH0 8 5
        DS>ATTACH RP0X RH0 DBA0(RP0X is, e.g. RP06)
        DS>SELECT DBA0

This is for drive 0 on mba0; use 9 instead of 8 for mba1, etc.

Formatting an RM0X. Follow the above procedures except that the ATTACH and SELECT lines should read:

        DS>ATTACH RH780 SBI RH0 8 5
        DS>ATTACH RM0X RH0 DRA0
        DS>SELECT DRA0

Don't forget to put your UNIX console floppy back in the floppy disk drive.

SEE ALSO

        bad144(8), badsect(8), newfs(8)

BUGS

An equivalent facility should be available which operates under a running UNIX system.

It should be possible to define more precisely what a "hard ECC" error is; e.g. the maximum unacceptable ECC width.

NAME
     fsck — file system consistency check and interactive repair

SYNOPSIS
     /etc/fsck —p [filesystem ...]
     /etc/fsck [—b block#] [—y] [—n] [filesystem] ...

DESCRIPTION
     The first form of *fsck* preens a standard set of filesystems or the specified file systems. It is
     normally used in the script /etc/rc during automatic reboot. In this case *fsck* reads the table
     /etc/fstab to determine which file systems to check. It uses the information there to inspect
     groups of disks in parallel taking maximum advantage of i/o overlap to check the file systems as
     quickly as possible. Normally, the root file system will be checked on pass 1, other "root"
     ("a" partition) file systems on pass 2, other small file systems on separate passes (e.g. the "d"
     file systems on pass 3 and the "e" file systems on pass 4), and finally the large user file systems
     on the last pass, e.g. pass 5. A pass number of 0 in fstab causes a disk to not be checked; simi-
     larly partitions which are not shown as to be mounted "rw" or "ro" are not checked.

     The system takes care that only a restricted class of innocuous inconsistencies can happen
     unless hardware or software failures intervene. These are limited to the following:

             Unreferenced inodes

             Link counts in inodes too large

             Missing blocks in the free list

             Blocks in the free list also in files

             Counts in the super-block wrong

     These are the only inconsistencies which *fsck* with the —p option will correct; if it encounters
     other inconsistencies, it exits with an abnormal return status and an automatic reboot will then
     fail. For each corrected inconsistency one or more lines will be printed identifying the file sys-
     tem on which the correction will take place, and the nature of the correction. After success-
     fully correcting a file system, *fsck* will print the number of files on that file system and the
     number of used and free blocks.

     Without the —p option, *fsck* audits and interactively repairs inconsistent conditions for file sys-
     tems. If the file system is inconsistent the operator is prompted for concurrence before each
     correction is attempted. It should be noted that a number of the corrective actions which are
     not fixable under the —p option will result in some loss of data. The amount and severity of
     data lost may be determined from the diagnostic output. The default action for each con-
     sistency correction is to wait for the operator to respond yes or no. If the operator does not
     have write permission *fsck* will default to a —n action.

     Fsck has more consistency checks than its predecessors *check, dcheck, fcheck,* and *icheck* com-
     bined.

     The following flags are interpreted by *fsck.*

     —b   Use the block specified immediately after the flag as the super block for the file system.
             Block 32 is always an alternate super block.

     —y   Assume a yes response to all questions asked by *fsck;* this should be used with great cau-
             tion as this is a free license to continue after essentially unlimited trouble has been
             encountered.

     —n   Assume a no response to all questions asked by *fsck;* do not open the file system for
             writing.

8

If no filesystems are given to *fsck* then a default list of file systems is read from the file /etc/fstab.

Inconsistencies checked are as follows:

1.   Blocks claimed by more than one inode or the free list.
2.   Blocks claimed by an inode or the free list outside the range of the file system.
3.   Incorrect link counts.
4.   Size checks:
            Directory size not of proper format.
5.   Bad inode format.
6.   Blocks not accounted for anywhere.
7.   Directory checks:
            File pointing to unallocated inode.
            Inode number out of range.
8.   Super Block checks:

            More blocks for inodes than there are in the file system.
9.   Bad free block list format.
10.   Total free block and/or free inode count incorrect.

Orphaned files and directories (allocated but unreferenced) are, with the operator's concurrence, reconnected by placing them in the **lost+found** directory. The name assigned is the inode number. The only restriction is that the directory **lost+found** must preexist in the root of the filesystem being checked and must have empty slots in which entries can be made. This is accomplished by making **lost+found**, copying a number of files to the directory, and then removing them (before *fsck* is executed).

Checking the raw device is almost always faster.

FILES
        /etc/fstab   contains default list of file systems to check.

DIAGNOSTICS
        The diagnostics produced by *fsck* are intended to be self-explanatory.

SEE ALSO
        fstab(5), fs(5), newfs(8), mkfs(8), crash(8V), reboot(8)

BUGS
        Inode numbers for . and .. in each directory should be checked for validity.

        There should be some way to start a **fsck** −p at pass *n.*

8

NAME

ftpd — DARPA Internet File Transfer Protocol server

SYNOPSIS

/etc/ftpd [−d] [−l] [−ttimeout]

DESCRIPTION

Ftpd is the DARPA Internet File Transfer Prototocol server process. The server uses the TCP protocol and listens at the port specified in the "ftp" service specification; see *services*(5).

If the −d option is specified, each socket created will have debugging turned on (SO.DEBUG). With debugging enabled, the system will trace all TCP packets sent and received on a socket. The program *trpt*(8C) may then be used to interpret the packet traces.

If the −l option is specified, each ftp session is logged on the standard output. This allows a line of the form '/etc/ftpd -l > /tmp/ftplog" to be used to conveniently maintain a log of ftp sessions.

The ftp server will timeout an inactive session after 60 seconds. If the −t option is specified, the inactivity timeout period will be set to *timeout*.

The ftp server currently supports the following ftp requests; case is not distinguished.

Request	Description
ACCT	specify account (ignored)
ALLO	allocate storage (vacuously)
APPE	append to a file
CWD	change working directory
DELE	delete a file
HELP	give help information
LIST	give list files in a directory ("ls -lg")
MODE	specify data transfer *mode*
NLST	give name list of files in directory ("ls")
NOOP	do nothing
PASS	specify password
PORT	specify data connection port
QUIT	terminate session
RETR	retrieve a file
RNFR	specify rename-from file name
RNTO	specify rename-to file name
STOR	store a file
STRU	specify data transfer *structure*
TYPE	specify data transfer *type*
USER	specify user name
XCUP	change to parent of current working directory
XCWD	change working directory
XMKD	make a directory
XPWD	print the current working directory
XRMD	remove a directory

The remaining ftp requests specified in Internet RFC 765 are recognized, but not implemented.

Ftpd interprets file names according to the "globbing" conventions used by *csh*(1). This allows users to utilize the metacharacters "*?[]{}~".

Ftpd authenticates users according to three rules.

1)   The user name must be in the password data base, */etc/passwd*, and not have a null password. In this case a password must be provided by the client before any file

8

operations may be performed.

2)   The user name must not appear in the file */etc/ftpusers*.

3)   If the user name is "anonymous" or "ftp", an anonymous ftp account must be present in the password file (user "ftp"). In this case the user is allowed to log in by specifying any password (by convention this is given as the client host's name).

In the last case, *ftpd* takes special measures to restrict the client's access privileges. The server performs a *chroot*(2) command to the home directory of the "ftp" user. In order that system security is not breached, it is recommended that the "ftp" subtree be constructed with care; the following rules are recommended.

~ftp)   Make the home directory owned by "ftp" and unwritable by anyone.

~ftp/bin)

Make this directory owned by the super-user and unwritable by anyone. The program *ls*(1) must be present to support the list commands. This program should have mode 111.

~ftp/etc)

Make this directory owned by the super-user and unwritable by anyone. The files *passwd*(5) and *group*(5) must be present for the *ls* command to work properly. These files should be mode 444.

~ftp/pub)

Make this directory mode 777 and owned by "ftp". Users should then place files which are to be accessible via the anonymous account in this directory.

SEE ALSO
ftp(1C),

BUGS
There is no support for aborting commands.

The anonymous account is inherently dangerous and should avoided when possible.

The server must run as the super-user to create sockets with privileged port numbers. It maintains an effective user id of the logged in user, reverting to the super-user only when binding addresses to sockets. The possible security holes have been extensively scrutinized, but are possibly incomplete.

NAME

gettable — get NIC format host tables from a host

SYNOPSIS

/etc/gettable *host*

DESCRIPTION

Gettable is a simple program used to obtain the NIC standard host tables from a "nicname" server. The indicated *host* is queried for the tables. The tables, if retrieved, are placed in the file *hosts.txt*.

Gettable operates by opening a TCP connection to the port indicated in the service specification for "nicname". A request is then made for "ALL" names and the resultant information is placed in the output file.

Gettable is best used in conjunction with the *htable*(8) program which converts the NIC standard file format to that used by the network library lookup routines.

SEE ALSO

intro(3N), htable(8)

BUGS

Should allow requests for only part of the database.

8

NAME

getty — set terminal mode

SYNOPSIS

/etc/getty [type]

DESCRIPTION

Getty is invoked by *init*(8) immediately after a terminal is opened, following the making of a connection. While reading the name *getty* attempts to adapt the system to the speed and type of terminal being used.

Init calls *getty* with an argument specified by the *ttys* file entry for the terminal line. The argument can be used to make *getty* treat the line specially. This argument is used as an index into the *gettytab*(5) database, to determine the characteristics of the line. If there is no argument, or there is no such table, the default table is used. If there is no /etc/gettytab a set of system defaults is used. If indicated by the table located, *getty* will clear the terminal screen, print a banner heading, and prompt for a login name. Usually either the banner of the login prompt will include the system hostname. Then the user's name is read, a character at a time. If a null character is received, it is assumed to be the result of the user pushing the 'break' ('interrupt') key. The speed is usually then changed and the 'login:' is typed again; a second 'break' changes the speed again and the 'login:' is typed once more. Successive 'break' characters cycle through the some standard set of speeds.

The user's name is terminated by a new-line or carriage-return character. The latter results in the system being set to treat carriage returns appropriately (see *tty*(4)).

The user's name is scanned to see if it contains any lower-case alphabetic characters; if not, and if the name is nonempty, the system is told to map any future upper-case characters into the corresponding lower-case characters.

Finally, login is called with the user's name as argument.

Most of the default actions of *getty* can be circumvented, or modified, by a suitable *gettytab* table.

Getty can be set to timeout after some interval, which will cause dial up lines to hang up if the login name is not entered reasonably quickly.

FILES

/etc/gettytab

SEE ALSO

gettytab(5), init(8), login(1), ioctl(2), tty(4), ttys(5).

BUGS

Currently, the format of /etc/ttys limits the permitted table names to a single character, this should be expanded.

/etc/ttys should be replaced completely.

8

NAME
      halt — stop the processor

SYNOPSIS
      /etc/halt [−n] [−q] [−y]

DESCRIPTION
      Halt writes out sandbagged information to the disks and then stops the processor. The machine does not reboot, even if the auto-reboot switch is set on the console.

      The −n option prevents the sync before stopping. The −q option causes a quick halt, no graceful shutdown is attempted. The −y option is needed if you are trying to halt the system from a dialup.

SEE ALSO
      reboot(8), shutdown(8)

BUGS
      It is very difficult to halt a VAX, as the machine wants to then reboot itself. A rather tight loop suffices.

8

NAME

htable — convert NIC standard format host tables

SYNOPSIS

/etc/htable [—c *connected-nets*] [—l *local-nets*] *file*

DESCRIPTION

Htable is used to convert host files in the format specified in Internet RFC 810 to the format used by the network library routines. Three files are created as a result of running *htable*: *hosts*, *networks*, and *gateways*. The *hosts* file is used by the *gethostent*(3N) routines in mapping host names to addresses. The *networks* file is used by the *getnetent*(3N) routines in mapping network names to numbers. The *gateways* file is used by the routing daemon in identifying "passive" Internet gateways; see *routed*(8C) for an explanation.

If any of the files *localhosts*, *localnetworks*, or *localgateways* are present in the current directory, the file's contents is prepended to the output file. Of these, only the gateways file is interpreted. This allows sites to maintain local aliases and entries which are not normally present in the master database. Only one gateway to each network will be placed in the gateways file; a gateway listed in the localgateways file will override any in the input file.

A list of networks to which the host is directly connected is specified with the —c flag. The networks, separated by commas, may be given by name or in internet-standard dot notation, e.g. —c arpanet,128.32,local-ether-net. *Htable* only includes gateways which are directly connected to one of the networks specified, or which can be reached from another gateway on a connected net.

If the —l option is given with a list of networks (in the same format as for —c), these networks will be treated as "local," and information about hosts on local networks is taken only from the localhosts file. Entries for local hosts from the main database will be omitted. This allows the localhosts file to completely override any entries in the input file.

Htable is best used in conjunction with the *gettable*(8C) program which retrieves the NIC database from a host.

SEE ALSO

intro(3N), gettable(8C)

NAME
icheck — file system storage consistency check

SYNOPSIS
/etc/icheck [−s] [−b numbers] [filesystem]

DESCRIPTION
N.B.: *Icheck* is obsoleted for normal consistency checking by *fsck*(8).

Icheck examines a file system, builds a bit map of used blocks, and compares this bit map against the free list maintained on the file system. If the file system is not specified, a set of default file systems is checked. The normal output of *icheck* includes a report of

The total number of files and the numbers of regular, directory, block special and character special files.

The total number of blocks in use and the numbers of single-, double-, and triple-indirect blocks and directory blocks.

The number of free blocks.

The number of blocks missing; i.e. not in any file nor in the free list.

The −s option causes *icheck* to ignore the actual free list and reconstruct a new one by rewriting the super-block of the file system. The file system should be dismounted while this is done; if this is not possible (for example if the root file system has to be salvaged) care should be taken that the system is quiescent and that it is rebooted immediately afterwards so that the old, bad in-core copy of the super-block will not continue to be used. Notice also that the words in the super-block which indicate the size of the free list and of the i-list are believed. If the super-block has been curdled these words will have to be patched. The −s option causes the normal output reports to be suppressed.

Following the −b option is a list of block numbers; whenever any of the named blocks turns up in a file, a diagnostic is produced.

Icheck is faster if the raw version of the special file is used, since it reads the i-list many blocks at a time.

FILES
Default file systems vary with installation.

SEE ALSO
fsck(8), dcheck(8), ncheck(8), fs(5), clri(8)

DIAGNOSTICS
For duplicate blocks and bad blocks (which lie outside the file system) *icheck* announces the difficulty, the i-number, and the kind of block involved. If a read error is encountered, the block number of the bad block is printed and *icheck* considers it to contain 0. 'Bad freeblock' means that a block number outside the available space was encountered in the free list. '*n* dups in free' means that *n* blocks were found in the free list which duplicate blocks either in some file or in the earlier part of the free list.

BUGS
Since *icheck* is inherently two-pass in nature, extraneous diagnostics may be produced if applied to active file systems.

It believes even preposterous super-blocks and consequently can get core images.

The system should be fixed so that the reboot after fixing the root file system is not necessary.

8

NAME
      ifconfig — configure network interface parameters

SYOPNSIS
      /etc/ifconfig interface [*address*] [*parameters*]

DESCRIPTION
      Ifconfig is used to assign an address to a network interface and/or configure network interface parameters. *Ifconfig* must be used at boot time to define the network address of each interface present on a machine; it may also be used at a later time to redefine an interface's address. The *interface* parameter is a string of the form "name unit", e.g. "en0", while the address is either a host name present in the host name data base, *hosts*(5), or a DARPA Internet address expressed in the Internet standard "dot notation".

      The following parameters may be set with *ifconfig*:

up   Mark an interface "up".

down   Mark an interface "down". When an interface is marked "down", the system will not attempt to transmit messages through that interface.

trailers   Enable the use of a "trailer" link level encapsulation when sending (default). If a network interface supports *trailers*, the system will, when possible, encapsulate outgoing messages in a manner which minimizes the number of memory to memory copy operations performed by the receiver.

−trailers   Disable the use of a "trailer" link level encapsulation.

arp   Enable the use of the Address Resolution Protocol in mapping between network level addresses and link level addresses (default). This is currently implemented for mapping between DARPA Internet addreses and 10Mb/s Ethernet addresses.

−arp   Disable the use of the Address Resolution Protocol.

      Ifconfig displays the current configuration for a network interface when no optional parameters are supplied.

      Only the super-user may modify the configuration of a network interface.

DIAGNOSTICS
      Messages indicating the specified interface does not exit, the requested address is unknown, the user is not privileged and tried to alter an interface's configuration.

SEE ALSO
      rc(8), intro(4N), netstat(1)

NAME
      implog — IMP log interpreter

SYNOPSIS
      /etc/implog [−D] [−f] [−c] [−l [link]] [−h host#] [−i imp#] [−t message-type]

DESCRIPTION
      Implog is program which interprets the message log produced by *implogd*(8C).

      If no arguments are specified, *implog* interprets and prints every message present in the message file. Options may be specified to force printing only a subset of the logged messages.

      −D   Do not show data messages.

      −f   Follow the logging process in action. This flags causes *implog* to print the current contents of the log file, then check for new logged messages every 5 seconds.

      −c   In addition to printing any data messages logged, show the contents of the data in hexadecimal bytes.

      −l [link#]
              Show only those messages received on the specified "link". If no value is given for the link, the link number of the IP protocol is assumed.

      −h host#
              Show only those messages received from the specified host. (Usually specified in conjunction with an imp.)

      −i imp#
              Show only those messages received from the specified imp.

      −t message-type
              Show only those messages received of the specified message type.

SEE ALSO
      imp(4P), implogd(8C)

BUGS
      Can not specify multiple hosts, imps, etc. Can not follow reception of messages without looking at those currently in the file.

8

NAME

implogd — IMP logger process

SYNOPSIS

/etc/implogd [−d]

DESCRIPTION

Implogd is program which logs messages from the IMP, placing them in the file */usr/adm/implog*.

Entries in the file are variable length. Each log entry has a fixed length header of the form:

```
struct sockstamp {
        short    sin_family;
        u_short sin_port;
        struct   in_addr sin_addr;
        time_t   sin_time;
        int      sin_len;
};
```

followed, possibly, by the message received from the IMP. Each time the logging process is started up it places a time stamp entry in the file (a header with *sin_len* field set to 0).

The logging process will catch only those message from the IMP which are not processed by a protocol module, e.g. IP. This implies the log should contain only status information such as "IMP going down" messages and, perhaps, stray NCP messages.

SEE ALSO

imp(4P), implog(8C)

BUGS

The messages should probably be sent to the system error logging process instead of maintaining yet another log file.

NAME

   init — process control initialization

SYNOPSIS

   /etc/init

DESCRIPTION

   Init is invoked inside UNIX as the last step in the boot procedure. It normally then runs the automatic reboot sequence as described in *reboot*(8), and if this succeeds, begins multi-user operation. If the reboot fails, it commences single user operation by giving the super-user a shell on the console. It is possible to pass parameters from the boot program to *init* so that single user operation is commenced immediately. When such single user operation is terminated by killing the single-user shell (i.e. by hitting ^D), *init* runs */etc/rc* without the reboot parameter. This command file performs housekeeping operations such as removing temporary files, mounting file systems, and starting daemons.

   In multi-user operation, *init's* role is to create a process for each terminal port on which a user may log in. To begin such operations, it reads the file */etc/ttys* and forks several times to create a process for each terminal specified in the file. Each of these processes opens the appropriate terminal for reading and writing. These channels thus receive file descriptors 0, 1 and 2, the standard input and output and the diagnostic output. Opening the terminal will usually involve a delay, since the *open* is not completed until someone is dialed up and carrier established on the channel. If a terminal exists but an error occurs when trying to open the terminal *init* complains by writing a message to the system console; the message is repeated every 10 minutes for each such terminal until the terminal is shut off in /etc/ttys and init notified (by a hangup, as described below), or the terminal becomes accessible (init checks again every minute). After an open succeeds, */etc/getty* is called with argument as specified by the second character of the *ttys* file line. *Getty* reads the user's name and invokes *login* to log in the user and execute the Shell.

   Ultimately the Shell will terminate because of an end-of-file either typed explicitly or generated as a result of hanging up. The main path of *init*, which has been waiting for such an event, wakes up and removes the appropriate entry from the file *utmp*, which records current users, and makes an entry in */usr/adm/wtmp*, which maintains a history of logins and logouts. The *wtmp* entry is made only if a user logged in successfully on the line. Then the appropriate terminal is reopened and *getty* is reinvoked.

   Init catches the *hangup* signal (signal SIGHUP) and interprets it to mean that the file */etc/ttys* should be read again. The Shell process on each line which used to be active in *ttys* but is no longer there is terminated; a new process is created for each added line; lines unchanged in the file are undisturbed. Thus it is possible to drop or add phone lines without rebooting the system by changing the *ttys* file and sending a *hangup* signal to the *init* process: use 'kill —HUP 1.'

   Init will terminate multi-user operations and resume single-user mode if sent a terminate (TERM) signal, i.e. "kill —TERM 1". If there are processes outstanding which are deadlocked (due to hardware or software failure), *init* will not wait for them all to die (which might take forever), but will time out after 30 seconds and print a warning message.

   Init will cease creating new *getty*'s and allow the system to slowly die away, if it is sent a terminal stop (TSTP) signal, i.e. "kill —TSTP 1". A later hangup will resume full multi-user operations, or a terminate will initiate a single user shell. This hook is used by *reboot*(8) and *halt*(8).

   Init's role is so critical that if it dies, the system will reboot itself automatically. If, at bootstrap time, the *init* process cannot be located, the system will loop in user mode at location 0x13.

DIAGNOSTICS

   init: *tty*: **cannot open.** A terminal which is turned on in the *rc* file cannot be opened, likely because the requisite lines are either not configured into the system or the associated device

8

was not attached during boot-time system configuration.

WARNING: Something is hung (wont die); ps axl advised. A process is hung and could not be killed when the system was shutting down. This is usually caused by a process which is stuck in a device driver due to a persistent device error condition.

FILES

/dev/console, /dev/tty*, /etc/utmp, /usr/adm/wtmp, /etc/ttys, /etc/rc

SEE ALSO

login(1), kill(1), sh(1), ttys(5), crash(8V), getty(8), rc(8), reboot(8), halt(8), shutdown(8)

NAME
kgmon — generate a dump of the operating system's profile buffers

SYNOPSIS
/etc/kgmon [−b] [−h] [−r] [−p] [system] [memory]

DESCRIPTION
Kgmon is a tool used when profiling the operating system. When no arguments are supplied, *kgmon* indicates the state of operating system profiling as running, off, or not configured. (see *config*(8)) If the −p flag is specified, *kgmon* extracts profile data from the operating system and produces a *gmon.out* file suitable for later analysis by *gprof*(1).

The following options may be specified:

−b   Resume the collection of profile data.

−h   Stop the collection of profile data.

−p   Dump the contents of the profile buffers into a *gmon.out* file.

−r   Reset all the profile buffers. If the −p flag is also specified, the *gmon.out* file is generated before the buffers are reset.

If neither −b nor −h is specified, the state of profiling collection remains unchanged. For example, if the −p flag is specified and profile data is being collected, profiling will be momentarily suspended, the operating system profile buffers will be dumped, and profiling will be immediately resumed.

FILES
/vmunix — the default system
/dev/kmem — the default memory

SEE ALSO
gprof(1), config(8)

DIAGNOSTICS
Users with only read permission on /dev/kmem cannot change the state of profiling collection. They can get a *gmon.out* file with the warning that the data may be inconsistent if profiling is in progress.

8

NAME

lpc — line printer control program

SYNOPSIS

/etc/lpc [command [argument ...]]

DESCRIPTION

Lpc is used by the system administrator to control the operation of the line printer system. For each line printer configured in /etc/printcap, *lpc* may be used to:

- disable or enable a printer,
- disable or enable a printer's spooling queue,
- rearrange the order of jobs in a spooling queue,
- find the status of printers, and their associated spooling queues and printer dameons.

Without any arguments, *lpc* will prompt for commands from the standard input. If arguments are supplied, *lpc* interprets the first argument as a command and the remaining arguments as parameters to the command. The standard input may be redirected causing *lpc* to read commands from file. Commands may be abreviated; the following is the list of recognized commands.

? [command ...]

help [command ...]
> Print a short description of each command specified in the argument list, or, if no arguments are given, a list of the recognized commands.

abort { all | printer ... }
> Terminate an active spooling daemon on the local host immediately and then disable printing (preventing new daemons from being started by *lpr*) for the specified printers.

clean { all | printer ... }
> Remove all files beginning with "cf", "tf", or "df" from the specified printer queue(s) on the local machine.

enable { all | printer ... }
> Enable spooling on the local queue for the listed printers. This will allow *lpr* to put new jobs in the spool queue.

exit

quit
> Exit from lpc.

disable { all | printer ... }
> Turn the specified printer queues off. This prevents new printer jobs from being entered into the queue by *lpr*.

restart { all | printer ... }
> Attempt to start a new printer daemon. This is useful when some abnormal condition causes the daemon to die unexpectedly leaving jobs in the queue. *Lpq* will report that there is no daemon present when this condition occurs.

start { all | printer ... }
> Enable printing and start a spooling daemon for the listed printers.

status [all] [printer ...]
> Display the status of daemons and queues on the local machine.

stop { all | printer ... }
> Stop a spooling daemon after the current job completes and disable printing.

topq printer [jobnum ...] [user ...]
        Place the jobs in the order listed at the top of the printer queue.

FILES
/etc/printcap	printer description file
/usr/spool/•	spool directories
/usr/spool/•/lock	lock file for queue control

SEE ALSO
    lpd(8), lpr(1), lpq(1), lprm(1), printcap(5)

DIAGNOSTICS
?Ambiguous command	abreviation matches more than one command
?Invalid command	no match was found
?Privileged command	command can be executed by root only

8

NAME

lpd — line printer daemon

SYNOPSIS

/usr/lib/lpd [-l] [-L logfile] [port #]

DESCRIPTION

Lpd is the line printer daemon (spool area handler) and is normally invoked at boot time from the *rc*(8) file. It makes a single pass through the *printcap*(5) file to find out about the existing printers and prints any files left after a crash. It then uses the system calls *listen*(2) and *accept*(2) to receive requests to print files in the queue, transfer files to the spooling area, display the queue, or remove jobs from the queue. In each case, it forks a child to handle the request so the parent can continue to listen for more requests. The Internet port number used to rendezvous with other processes is normally obtained with *getservbyname*(3) but can be changed with the *port#* argument. The —L option changes the file used for writing error conditions from the system console to *logfile*. The —l flag causes *lpd* to log valid requests received from the network. This can be useful for debugging purposes.

Access control is provided by two means. First, All requests must come from one of the machines listed in the file */etc/hosts.equiv*. Second, if the "rs" capability is specified in the *printcap* entry for the printer being accessed, *lpr* requests will only be honored for those users with accounts on the machine with the printer.

The file *lock* in each spool directory is used to prevent multiple daemons from becoming active simultaneously, and to store information about the daemon process for *lpr*(1), *lpq*(1), and *lprm*(1). After the daemon has successfully set the lock, it scans the directory for files beginning with *cf*. Lines in each *cf* file specify files to be printed or non-printing actions to be performed. Each such line begins with a key character to specify what to do with the remainder of the line.

J	Job Name. String to be used for the job name on the burst page.
C	Classification. String to be used for the classification line on the burst page.
L	Literal. The line contains identification info from the password file and causes the banner page to be printed.
T	Title. String to be used as the title for *pr*(1).
H	Host Name. Name of the machine where *lpr* was invoked.
P	Person. Login name of the person who invoked *lpr*. This is used to verify ownership by *lprm*.
M	Send mail to the specified user when the current print job completes.
f	Formatted File. Name of a file to print which is already formatted.
l	Like "f" but passes control characters and does not make page breaks.
p	Name of a file to print using *pr*(1) as a filter.
t	Troff File. The file contains *troff*(1) output (cat phototypesetter commands).
d	DVI File. The file contains *Tex*(l) output (DVI format from Standford).
g	Graph File. The file contains data produced by *plot*(3X).
c	Cifplot File. The file contains data produced by *cifplot*.
v	The file contains a raster image.
r	The file contains text data with FORTRAN carriage control characters.
1	Troff Font R. Name of the font file to use instead of the default.

2	Troff Font I. Name of the font file to use instead of the default.
3	Troff Font B. Name of the font file to use instead of the default.
4	Troff Font S. Name of the font file to use instead of the default.
W	Width. Changes the page width (in characters) used by *pr*(1) and the text filters.
I	Indent. The number of characters to indent the output by (in ascii).
U	Unlink. Name of file to remove upon completion of printing.
N	File name. The name of the file which is being printed, or a blank for the standard input (when *lpr* is invoked in a pipeline).

If a file can not be opened, a message will be placed in the log file (normally the console). *Lpd* will try up to 20 times to reopen a file it expects to be there, after which it will skip the file to be printed.

Lpd uses *flock*(2) to provide exclusive access to the lock file and to prevent multiple deamons from becoming active simultaneously. If the daemon should be killed or die unexpectedly, the lock file need not be removed. The lock file is kept in a readable ASCII form and contains two lines. The first is the process id of the daemon and the second is the control file name of the current job being printed. The second line is updated to reflect the current status of *lpd* for the programs *lpq*(1) and *lprm*(1).

FILES

/etc/printcap	printer description file
/usr/spool/•	spool directories
/dev/lp•	line printer devices
/dev/printer	socket for local requests
/etc/hosts.equiv	lists machine names allowed printer access

SEE ALSO

lpc(8), pac(1), lpr(1), lpq(1), lprm(1), printcap(5)
4.2BSD Line Printer Spooler Manual

8

NAME

makedev — make system special files

SYNOPSIS

/dev/MAKEDEV *device...*

DESCRIPTION

MAKEDEV is a shell script normally used to install special files. It resides in the */dev* directory, as this is the normal location of special files. Arguments to *MAKEDEV* are usually of the form *device-name?* where *device-name* is one of the supported devices listed in section 4 of the manual and "?" is a logical unit number (0-9). A few special arguments create assorted collections of devices and are listed below.

std   Create the *standard* devices for the system; e.g. /dev/console, /dev/tty. The VAX-11/780 console floppy device, /dev/floppy, and VAX-11/750 and VAX-11/730 console cassette device(s), /dev/tu?, are also created with this entry.

local   Create those devices specific to the local site. This request causes the shell file */dev/MAKEDEV.local* to be executed. Site specific commands, such as those used to setup dialup lines as "ttyd?" should be included in this file.

Since all devices are created using *mknod*(8), this shell script is useful only to the super-user.

DIAGNOSTICS

Either self-explanatory, or generated by one of the programs called from the script. Use "sh -x MAKEDEV" in case of trouble.

SEE ALSO

intro(4), config(8), mknod(8)

BUGS

When more than one piece of hardware of the same "kind" is present on a machine (for instance, a dh and a dmf), naming conflicts arise.

NAME

makekey — generate encryption key

SYNOPSIS

/usr/lib/makekey

DESCRIPTION

Makekey improves the usefulness of encryption schemes depending on a key by increasing the amount of time required to search the key space. It reads 10 bytes from its standard input, and writes 13 bytes on its standard output. The output depends on the input in a way intended to be difficult to compute (that is, to require a substantial fraction of a second).

The first eight input bytes (the *input key*) can be arbitrary ASCII characters. The last two (the *salt*) are best chosen from the set of digits, upper- and lower-case letters, and '.' and '/'. The salt characters are repeated as the first two characters of the output. The remaining 11 output characters are chosen from the same set as the salt and constitute the *output key*.

The transformation performed is essentially the following: the salt is used to select one of 4096 cryptographic machines all based on the National Bureau of Standards DES algorithm, but modified in 4096 different ways. Using the input key as key, a constant string is fed into the machine and recirculated a number of times. The 64 bits that come out are distributed into the 66 useful key bits in the result.

Makekey is intended for programs that perform encryption (for instance, *ed* and *crypt*(1)). Usually makekey's input and output will be pipes.

SEE ALSO

crypt(1), ed(1)

8

NAME
mkfs — construct a file system

SYNOPSIS
/etc/mkfs special size [nsect] [ntrack] [blksize] [fragsize] [ncpg] [minfree] [rps]

DESCRIPTION
N.B.: file system are normally created with the *newfs*(8) command.

Mkfs constructs a file system by writing on the special file *special*. The numeric size specifies the number of sectors in the file system. *Mkfs* builds a file system with a root directory and a *lost+found* directory. (see *fsck*(8)) The number of i-nodes is calculated as a function of the file system size. No boot program is initialized by *mkfs* (see *newfs*(8).)

The optional arguments allow fine tune control over the parameters of the file system. **Nsect** specify the number of sectors per track on the disk. **Ntrack** specify the number of tracks per cylinder on the disk. **Blksize** gives the primary block size for files on the file system. It must be a power of two, currently selected from 4096 or 8192. **Fragsize** gives the fragment size for files on the file system. The **fragsize** represents the smallest amount of disk space that will be allocated to a file. It must be a power of two currently selected from the range 512 to 8192. **Ncpg** specifies the number of disk cylinders per cylinder group. This number must be in the range 1 to 32. **Minfree** specifies the minimum percentage of free disk space allowed. Once the file system capacity reaches this threshold, only the super-user is allowed to allocate disk blocks. The default value is 10%. If a disk does not revolve at 60 revolutions per second, the **rps** parameter may be specified. Users with special demands for their file systems are referred to the paper cited below for a discussion of the tradeoffs in using different configurations.

SEE ALSO
fs(5), dir(5), fsck(8), newfs(8), tunefs(8)

McKusick, Joy, Leffler; "A Fast File System for Unix", Computer Systems Research Group, Dept of EECS, Berkeley, CA 94720; TR #7, September 1982.

BUGS
There should be some way to specify bad blocks.

8

NAME

  mklost+found — make a lost+found directory for fsck

SYNOPSIS

  /etc/mklost+found

DESCRIPTION

  A directory *lost+found* is created in the current directory and a number of empty files are created therein and then removed so that there will be empty slots for *fsck*(8). This command should not normally be needed since *mkfs*(8) automatically creates the *lost+found* directory when a new file system is created.

SEE ALSO

  fsck(8), mkfs(8)

8

NAME

      mknod — build special file

SYNOPSIS

      /etc/mknod name [**c**] [**b**] major minor

DESCRIPTION

      Mknod makes a special file. The first argument is the *name* of the entry. The second is **b** if the special file is block-type (disks, tape) or **c** if it is character-type (other devices). The last two arguments are numbers specifying the *major* device type and the *minor* device (e.g. unit, drive, or line number).

      The assignment of major device numbers is specific to each system. They have to be dug out of the system source file *conf.c*.

SEE ALSO

      mknod(2)

NAME

    mkproto — construct a prototype file system

SYNOPSIS

    /etc/mkproto special proto

DESCRIPTION

    Mkproto is used to bootstrap a new file system. First a new file system is created using *newfs*(8). *Mkproto* is then used to copy files from the old file system into the new file system according to the directions found in the prototype file *proto*. The prototype file contains tokens separated by spaces or new lines. The first tokens comprise the specification for the root directory. File specifications consist of tokens giving the mode, the user-id, the group id, and the initial contents of the file. The syntax of the contents field depends on the mode.

    The mode token for a file is a 6 character string. The first character specifies the type of the file. (The characters —bcd specify regular, block special, character special and directory files respectively.) The second character of the type is either u or — to specify set-user-id mode or not. The third is g or — for the set-group-id mode. The rest of the mode is a three digit octal number giving the owner, group, and other read, write, execute permissions, see *chmod*(1).

    Two decimal number tokens come after the mode; they specify the user and group ID's of the owner of the file.

    If the file is a regular file, the next token is a pathname whence the contents and size are copied.

    If the file is a block or character special file, two decimal number tokens follow which give the major and minor device numbers.

    If the file is a directory, *mkproto* makes the entries . and .. and then reads a list of names and (recursively) file specifications for the entries in the directory. The scan is terminated with the token $.

    A sample prototype specification follows:

```
d——777 3 1
usr     d——777 3 1
        sh      ———755 3 1 /bin/sh
        ken     d——755 6 1
                $
        b0      b——644 3 1 0 0
        c0      c——644 3 1 0 0
                $
    $
```

SEE ALSO

    fs(5), dir(5), fsck(8), newfs(8)

BUGS

    There should be some way to specify links.

    There should be some way to specify bad blocks.

    Mkproto can only be run on virgin file systems. It should be possible to copy files into existent file systems.

8

NAME
        mount, umount — mount and dismount file system

SYNOPSIS
        /etc/mount [special name [−r]]

        /etc/mount −a

        /etc/umount special

        /etc/umount −a

DESCRIPTION
        Mount announces to the system that a removable file system is present on the device *special*.
        The file *name* must exist already; it must be a directory (unless the root of the mounted file
        system is not a directory). It becomes the name of the newly mounted root. The optional
        argument −r indicates that the file system is to be mounted read-only.

        Umount announces to the system that the removable file system previously mounted on device
        special is to be removed.

        If the −a option is present for either *mount* or *umount,* all of the file systems described in
        /etc/fstab are attempted to be mounted or unmounted. In this case, *special* and *name* are taken
        from */etc/fstab*. The *special* file name from */etc/fstab* is the block special name.

        These commands maintain a table of mounted devices in */etc/mtab*. If invoked without an argu-
        ment, *mount* prints the table.

        Physically write-protected and magnetic tape file systems must be mounted read-only or errors
        will occur when access times are updated, whether or not any explicit write is attempted.

FILES
        /etc/mtab   mount table
        /etc/fstab   file system table

SEE ALSO
        mount(2), mtab(5), fstab(5)

BUGS
        Mounting file systems full of garbage will crash the system.
        Mounting a root directory on a non-directory makes some apparently good pathnames invalid.

8

NAME

    ncheck — generate names from i-numbers

SYNOPSIS

    /etc/ncheck [−i numbers] [−a] [−s] [filesystem]

DESCRIPTION

    N.B.: For most normal file system maintenance, the function of *ncheck* is subsumed by *fsck*(8).

    Ncheck with no argument generates a pathname vs. i-number list of all files on a set of default file systems. Names of directory files are followed by '/.'. The −i option reduces the report to only those files whose i-numbers follow. The −a option allows printing of the names '.' and '..', which are ordinarily suppressed. The −s option reduces the report to special files and files with set-user-ID mode; it is intended to discover concealed violations of security policy.

    A file system may be specified.

    The report is in no useful order, and probably should be sorted.

SEE ALSO

    sort(1), dcheck(8), fsck(8), icheck(8)

DIAGNOSTICS

    When the filesystem structure is improper, '??' denotes the 'parent' of a parentless file and a pathname beginning with '...' denotes a loop.

NAME

newfs — construct a new file system

SYNOPSIS

/etc/newfs [−v] [−n] [mkfs-options] special disk-type

DESCRIPTION

Newfs is a "friendly" front-end to the *mkfs*(8) program. *Newfs* will look up the type of disk a file system is being created on in the disk description file *letcldisktab*, calculate the appropriate parameters to use in calling *mkfs*, then build the file system by forking *mkfs* and, if the file system is a root partition, install the necessary bootstrap programs in the initial 8 sectors of the device. The −n option prevents the bootstrap programs from being installed.

If the −v option is supplied, *newfs* will print out its actions, including the parameters passed to *mkfs*.

Options which may be used to override default parameters passed to *mkfs* are:

−s size   The size of the file system in sectors.

−b block-size
The block size of the file system in bytes.

−f frag-size
The fragment size of the file system in bytes.

−t #tracks/cylinder

−c #cylinders/group
The number of cylinders per cylinder group in a file system. The default value used is 16.

−m free space %
The percentage of space reserved from normal users; the minimum free space threshhold. The default value used is 10%.

−r revolutions/minute
The speed of the disk in revolutions per minute (normally 3600).

−S sector-size
The size of a sector in bytes (almost never anything but 512).

−i number of bytes per inode
This specifies the density of inodes in the file system. The default is to create an inode for each 2048 bytes of data space. If fewer inodes are desired, a larger number should be used; to create more inodes a smaller number should be given.

FILES

/etc/disktab   for disk geometry and file system partition information
/etc/mkfs   to actually build the file system
/usr/mdec   for boot strapping programs

SEE ALSO

disktab(5), fs(5), diskpart(8), fsck(8), format(8), mkfs(8), tunefs(8)

McKusick, Joy, Leffler; "A Fast File System for Unix", Computer Systems Research Group, Dept of EECS, Berkeley, CA 94720; TR #7, September 1982.

BUGS

Should figure out the type of the disk without the user's help.

NAME
      pac — printer/ploter accounting information

SYNOPSIS
      /etc/pac [−Pprinter] [−pprice] [−s] [−r] [−c] [name ...]

DESCRIPTION
      Pac reads the printer/plotter accounting files, accumulating the number of pages (the usual
      case) or feet (for raster devices) of paper consumed by each user, and printing out how much
      each user consumed in pages or feet and dollars. If any *names* are specified, then statistics are
      only printed for those users; usually, statistics are printed for every user who has used any
      paper.

      The −P flag causes accounting to be done for the named printer. Normally, accounting is done
      for the default printer (site dependent) or the value of the environment variable **PRINTER** is
      used.

      The −p flag causes the value *price* to be used for the cost in dollars instead of the default value
      of 0.02.

      The −c flag causes the output to be sorted by cost; usually the output is sorted alphabetically
      by name.

      The −r flag reverses the sorting order.

      The −s flag causes the accounting information to be summarized on the summary accounting
      file; this summarization is necessary since on a busy system, the accounting file can grow by
      several lines per day.

FILES
      /usr/adm/?acct   raw accounting files
      /usr/adm/?_sum   summary accounting files

BUGS
      The relationship between the computed price and reality is as yet unknown.

8

NAME
     pstat — print system facts

SYNOPSIS
     /etc/pstat —aixptufT [suboptions] [system] [corefile]

DESCRIPTION
     Pstat interprets the contents of certain system tables. If *corefile* is given, the tables are sought
     there, otherwise in */dev/kmem*. The required namelist is taken from */vmunix* unless *system* is
     specified. Options are

     —a   Under —p, describe all process slots rather than just active ones.

     —i   Print the inode table with the these headings:

     LOC   The core location of this table entry.
     FLAGS   Miscellaneous state variables encoded thus:
              L   locked
              U   update time (*fs*(5)) must be corrected
              A   access time must be corrected
              M   file system is mounted here
              W   wanted by another process (L flag is on)
              T   contains a text file
              C   changed time must be corrected
              S   shared lock applied
              E   exclusive lock applied
              Z   someone waiting for an exclusive lock
     CNT   Number of open file table entries for this inode.
     DEV   Major and minor device number of file system in which this inode resides.
     RDC   Reference count of shared locks on the inode.
     WRC   Reference count of exclusive locks on the inode (this may be > 1 if, for example, a
              file descriptor is inherited across a fork).
     INO   I-number within the device.
     MODE   Mode bits, see *chmod*(2).
     NLK   Number of links to this inode.
     UID   User ID of owner.
     SIZ/DEV
              Number of bytes in an ordinary file, or major and minor device of special file.

     —x   Print the text table with these headings:

     LOC   The core location of this table entry.
     FLAGS   Miscellaneous state variables encoded thus:
              T   *ptrace*(2) in effect
              W   text not yet written on swap device
              L   loading in progress
              K   locked
              w   wanted (L flag is on)
              P   resulted from demand-page-from-inode exec format (see *execve*(2))

     DADDR   Disk address in swap, measured in multiples of 512 bytes.

     CADDR   Head of a linked list of loaded processes using this text segment.

     SIZE   Size of text segment, measured in multiples of 512 bytes.

     IPTR   Core location of corresponding inode.

     CNT   Number of processes using this text segment.

     CCNT   Number of processes in core using this text segment.

8

−p   Print process table for active processes with these headings:

LOC   The core location of this table entry.

S   Run state encoded thus:

    0   no process
    1   waiting for some event
    3   runnable
    4   being created
    5   being terminated
    6   stopped under trace

F   Miscellaneous state variables, or-ed together (hexadecimal):

    000001   loaded
    000002   the scheduler process
    000004   locked for swap out
    000008   swapped out
    000010   traced
    000020   used in tracing
    000080   in page-wait
    000100   prevented from swapping during *fork*(2)
    000200   gathering pages for raw i/o
    000400   exiting
    001000   process resulted from a *vfork*(2) which is not yet complete
    002000   another flag for *vfork*(2)
    004000   process has no virtual memory, as it is a parent in the context of *vfork*(2)
    008000   process is demand paging data pages from its text inode.
    010000   process has advised of anomalous behavior with *vadvise*(2).
    020000   process has advised of sequential behavior with *vadvise*(2).
    040000   process is in a sleep which will timeout.
    080000   a parent of this process has exited and this process is now considered detached.
    100000   process used 4.1BSD compatibility mode signal primitives, no system calls will restart.
    200000   process is owed a profiling tick.

POIP   number of pages currently being pushed out from this process.

PRI   Scheduling priority, see *setpriority*(2).

SIGNAL Signals received (signals 1-32 coded in bits 0-31),

UID   Real user ID.

SLP   Amount of time process has been blocked.

TIM   Time resident in seconds; times over 127 coded as 127.

CPU   Weighted integral of CPU time, for scheduler.

NI   Nice level, see *setpriority*(2).

PGRP   Process number of root of process group (the opener of the controlling terminal).

PID   The process ID number.

PPID   The process ID of parent process.

ADDR   If in core, the page frame number of the first page of the 'u-area' of the process. If swapped out, the position in the swap area measured in multiples of 512 bytes.

RSS   Resident set size − the number of physical page frames allocated to this process.

SRSS   RSS at last swap (0 if never swapped).

SIZE   Virtual size of process image (data+stack) in multiples of 512 bytes.

WCHAN   Wait channel number of a waiting process.

LINK   Link pointer in list of runnable processes.

TEXTP   If text is pure, pointer to location of text table entry.

CLKT   Countdown for real interval timer, *setitimer*(2) measured in clock ticks (10

8

milliseconds).

—t   Print table for terminals with these headings:

RAW   Number of characters in raw input queue.
CAN   Number of characters in canonicalized input queue.
OUT   Number of characters in putput queue.
MODE   See *tty*(4).
ADDR   Physical device address.
DEL   Number of delimiters (newlines) in canonicalized input queue.
COL   Calculated column position of terminal.
STATE Miscellaneous state variables encoded thus:
       W   waiting for open to complete
       O   open
       S   has special (output) start routine
       C   carrier is on
       B   busy doing output
       A   process is awaiting output
       X   open for exclusive use
       H   hangup on close
PGRP   Process group for which this is controlling terminal.
DISC   Line discipline; blank is old tty OTTYDISC or "new tty" for NTTYDISC or "net"
       for NETLDISC (see *bk*(4)).

—u   print information about a user process; the next argument is its address as given by
      ps(1). The process must be in main memory, or the file used can be a core image
      and the address 0.

—f   Print the open file table with these headings:

LOC   The core location of this table entry.

TYPE   The type of object the file table entry points to.
FLG   Miscellaneous state variables encoded thus:
       R   open for reading
       W   open for writing
       A   open for appending
CNT   Number of processes that know this open file.
INO   The location of the inode table entry for this file.
OFFS/SOCK
       The file offset (see *lseek*(2)), or the core address of the associated socket structure.

—s print information about swap space usage: the number of (1k byte) pages used and free is
given as well as the number of used pages which belong to text images.

—T prints the number of used and free slots in the several system tables and is useful for
checking to see how full system tables have become if the system is under heavy load.

FILES
      /vmunix   namelist
      /dev/kmem   default source of tables

SEE ALSO
      ps(1), stat(2), fs(5)
      K. Thompson, *UNIX Implementation*

BUGS
      It would be very useful if the system recorded "maximum occupancy" on the tables reported
      by —T; even more useful if these tables were dynamically allocated.

NAME

    quot — summarize file system ownership

SYNOPSIS

    /etc/quot [option] ... [filesystem]

DESCRIPTION

    Quot prints the number of blocks in the named *filesystem* currently owned by each user. If no *filesystem* is named, a default name is assumed. The following options are available:

    −n   Cause the pipeline ncheck filesystem | sort +0n | quot −n filesystem to produce a list of all files and their owners.

    −c   Print three columns giving file size in blocks, number of files of that size, and cumulative total of blocks in that size or smaller file.

    −f   Print count of number of files as well as space owned by each user.

FILES

    Default file system varies with system.
    /etc/passwd   to get user names

SEE ALSO

    ls(1), du(1)

8

NAME

    quotacheck — file system quota consistency checker

SYNOPSIS

    /etc/quotacheck [−v] filesystem...

    /etc/quotacheck [−v] −a

DESCRIPTION

    Quotacheck examines each file system, builds a table of current disc usage, and compares this table against that stored in the disc quota file for the file system. If any inconsistencies are detected, both the quota file and the current system copy of the incorrect quotas are updated (the latter only occurs if an active file system is checked).

    If the −a flag is supplied in place of any file system names, *quotacheck* will check all the file systems indicated in */etc/fstab* to be read-write with disc quotas.

    Normally *quotacheck* reports only those quotas modified. If the −v option is supplied, *quotacheck* will indicate the calculated disc quotas for each user on a particular file system.

    Quotacheck expects each file system to be checked to have a quota file named *quotas* in the root directory. If none is present, *quotacheck* will ignore the file system.

    Quotacheck is normally run at boot time from the */etc/rc.local* file, see *rc*(8), before enabling disc quotas with *quotaon*(8).

    Quotacheck accesses the raw device in calculating the actual disc usage for each user. Thus, the file systems checked should be quiescent while *quotacheck* is running.

FILES

    /etc/fstab   default file systems

SEE ALSO

    quota(2), setquota(2), quotaon(8)

NAME
     quotaon, quotaoff — turn file system quotas on and off

SYNOPSIS
     /etc/quotaon [−v] *filsys...*

     /etc/quotaon [−v] −a

     /etc/quotaoff [−v] *filsys...*

     /etc/quotaoff [−v] −a

DESCRIPTION
     Quotaon announces to the system that disc quotas should be enabled on one or more file systems. The file systems specified must have entries in /etc/fstab and be mounted at the time. The file system quota files must be present in the root directory of the specified file system and be named *quotas*. The optional argument −v causes *quotaon* to print a message for each file system where quotas are turned on. If, instead of a list of file systems, a −a argument is give to *quotaon*, all file systems in /etc/fstab marked read-write with quotas will have their quotas turned on. This is normally used at boot time to enable quotas.

     Quotaoff announces to the system that file systems specified should have any disc quotas turned off. As above, the −v forces a verbose message for each file system affected; and the −a option forces all file systems in /etc/fstab to have their quotas disabled.

     These commands update the status field of devices located in *letc/mtab* to indicate when quotas are on or off for each file system.

FILES
     /etc/mtab   mount table
     /etc/fstab   file system table

SEE ALSO
     setquota(2), mtab(5), fstab(5)

8

NAME

    rc — command script for auto-reboot and daemons

SYNOPSIS

    /etc/rc
    /etc/rc.local

DESCRIPTION

Rc is the command script which controls the automatic reboot and *rc.local* is the script holding commands which are pertinent only to a specific site.

When an automatic reboot is in progress, *rc* is invoked with the argument *autoboot* and runs a *fsck* with option —p to "preen" all the disks of minor inconsistencies resulting from the last system shutdown and to check for serious inconsistencies caused by hardware or software failure. If this auto-check and repair succeeds, then the second part of *rc* is run.

The second part of *rc,* which is run after a auto-reboot succeeds and also if *rc* is invoked when a single user shell terminates (see *init*(8)), starts all the daemons on the system, preserves editor files and clears the scratch directory /tmp. *Rc.local* is executed immediately before any other commands after a successful *fsck*. Normally, the first commands placed in the *rc.local* file define the machine's name, using *hostname*(1), and save any possible core image that might have been generated as a result of a system crash, *savecore*(8). The latter command is included in the *rc.local* file because the directory in which core dumps are saved is usually site specific.

SEE ALSO

    init(8), reboot(8), savecore(8)

BUGS

NAME
    rdump — file system dump across the network

SYNOPSIS
    /etc/rdump [key [*argument* ...] filesystem]

DESCRIPTION
    Rdump copies to magnetic tape all files changed after a certain date in the *filesystem*. The command is identical in operation to *dump*(8) except the *f* key should be specified and the file supplied should be of the form *machine:device*.

    Rdump creates a remote server, */etc/rmt*, on the client machine to access the tape device.

SEE ALSO
    dump(8), rmt(8C)

DIAGNOSTICS
    Same as *dump*(8) with a few extra related to the network.

8

NAME
        reboot — UNIX bootstrapping procedures

SYNOPSIS
        /etc/reboot [−n] [−q]

DESCRIPTION
        UNIX is started by placing it in memory at location zero and transferring to zero. Since the
        system is not reenterable, it is necessary to read it in from disk or tape each time it is to be
        bootstrapped.

        Rebooting a running system. When a UNIX is running and a reboot is desired, *shutdown*(8) is
        normally used. If there are no users then /etc/reboot can be used. Reboot causes the disks to
        be synced, and then a multi-user reboot (as described below) is initiated. This causes a system
        to be booted and an automatic disk check to be performed. If all this succeeds without
        incident, the system is then brought up for many users.

        Options to reboot are:

        −n   option avoids the sync. It can be used if a disk or the processor is on fire.

        −q   reboots quickly and ungracefully, without shutting down running processes first.

        Power fail and crash recovery. Normally, the system will reboot itself at power-up or after
        crashes. Provided the auto-restart is enabled on the machine front panel, an automatic con-
        sistency check of the file systems will be performed then and unless this fails the system will
        resume multi-user operations.

        Cold starts. These are processor type dependent. On an 11/780, there are two floppy files for
        each disk controller, both of which cause boots from unit 0 of the root file system of a con-
        troller located on mba0 or uba0. One gives a single user shell, while the other invokes the
        multi-user automatic reboot. Thus these files are HPS and HPM for the single and multi-user
        boot from MASSBUS RP06/RM03/RM05 disks, UPS and UPM for UNIBUS storage module
        controller and disks such as the EMULEX SC-21 and AMPEX 9300 pair, or HKS and HKM for
        RK07 disks.

        Giving the command

                >>>BOOT HPM

        Would boot the system from (e.g.) an RP06 and run the automatic consistency check as
        described in *fsck*(8). (Note that it may be necessary to type control-P to gain the attention of
        the LSI-11 before getting the >>> prompt.) The command

                >>>BOOT ANY

        invokes a version of the boot program in a way which allows you to specify any system as the
        system to be booted. It reads from the console a device specification (see below) followed
        immediately by a pathname.

        On an 11/750, the reset button will boot from the device selected by the front panel boot dev-
        ice switch. In systems with RK07's, position B normally selects the RK07 for boot. This will
        boot multi-user. To boot from RK07 with boot flags you may specify

                >>>B/*n* DMA0

        where, giving a *n* of 1 causes the boot program to ask for the name of the system to be
        bootstrapped, giving a *n* of 2 causes the boot program to come up single user, and a *n* of 3
        causes both of these actions to occur.

        The 11/750 boot procedure uses the boot roms to load block 0 off of the specified device. The
        /usr/mdec directory contains a number of bootstrap programs for the various disks which
        should be placed in a new pack automatically by *newfs*(8) when the "a" partition file system on

8

the pack is created.

On both processors, the *boot* program finds the corresponding file on the given device, loads that file into memory location zero, and starts the program at the entry address specified in the program header (after clearing off the high bit of the specified entry address.) Normal line editing characters can be used in specifying the pathname.

If you have a MASSBUS disk and wish to boot off of a file system which starts at cylinder 0 of unit 0, you can type "hp(0,0)vmunix" to the boot prompt; "up(0,0)vmunix" would specify a UNIBUS drive, "hk(0,0)vmunix" would specify an RK07 disk drive, "ra(0,0)vmunix" would specify a UDA50 disk drive, and "rb(0,0)vmunix" would specify a disk on a 730 IDC.

A device specification has the following form:

       device(unit, minor)

where *device* is the type of the device to be searched, *unit* is 8* the mba or uba number plus the unit number of the device, and *minor* is the minor device index. The following list of supported devices may vary from installation to installation:

hp	MASSBUS disk drive
up	UNIBUS storage module drive
ht	TE16,TU45,TU77 on MASSBUS
mt	TU78 on MASSBUS
hk	RK07 on UNIBUS
ra	storage module on a UDA50
rb	storage module on a 730 IDC
rl	RL02 on UNIBUS
tm	TM11 emulation tape drives on UNIBUS
ts	TS11 on UNIBUS
ut	UNIBUS TU45 emulator

For tapes, the minor device number gives a file offset.

In an emergency, the bootstrap methods described in the paper "Installing and Operating 4.2bsd" can be used to boot from a distribution tape.

FILES
/vmunix	system code
/boot	system bootstrap

SEE ALSO
       crash(8V), fsck(8), init(8), rc(8), shutdown(8), halt(8), newfs(8)

8

NAME

renice — alter priority of running processes

SYNOPSIS

/etc/renice priority [[−p] pid ...] [[−g] pgrp ...] [[−u] user ...]

DESCRIPTION

Renice alters the scheduling priority of one or more running processes. The *who* parameters are interpreted as process ID's, process group ID's, or user names. *Renice*'ing a process group causes all processes in the process group to have their scheduling priority altered. *Renice*'ing a user causes all processes owned by the user to have their scheduling priority altered. By default, the processes to be affected are specified by their process ID's. To force *who* parameters to be interpreted as process group ID's, a −g may be specified. To force the *who* parameters to be interpreted as user names, a −u may be given. Supplying −p will reset *who* interpretation to be (the default) process ID's. For example,

    /etc/renice +1 987 -u daemon root -p 32

would change the priority of process ID's 987 and 32, and all processes owned by users daemon and root.

Users other than the super-user may only alter the priority of processes they own, and can only monotonically increase their "nice value" within the range 0 to PRIO_MIN (20). (This prevents overriding administrative fiats.) The super-user may alter the priority of any process and set the priority to any value in the range PRIO_MAX (−20) to PRIO_MIN. Useful priorities are: 19 (the affected processes will run only when nothing else in the system wants to), 0 (the "base" scheduling priority), anything negative (to make things go very fast).

FILES

/etc/passwd to map user names to user ID's

SEE ALSO

getpriority(2), setpriority(2)

BUGS

If you make the priority very negative, then the process cannot be interrupted. To regain control you make the priority greater than zero. Non super-users can not increase scheduling priorities of their own processes, even if they were the ones that decreased the priorities in the first place.

8

NAME
        repquota — summarize quotas for a file system

SYNOPSIS
        repquota *filesys...*

DESCRIPTION
        Repquota prints a summary of the disc usage and quotas for the specified file systems. For each
        user the current number files and amount of space (in kilobytes) is printed, along with any
        quotas created with *edquota* (8).

        Only the super-user may view quotas which are not their own.

FILES
        quotas   at the root of each file system with quotas
        /etc/fstab   for file system names and locations

SEE ALSO
        quota(1), quota(2), quotacheck(8), quotaon(8), edquota(8)

DIAGNOSTICS
        Various messages about inaccessible files; self-explanatory.

8

NAME
     restore — incremental file system restore

SYNOPSIS
     /etc/restore key [name ...]

DESCRIPTION
     Restore reads tapes dumped with the *dump*(8) command. Its actions are controlled by the *key* argument. The *key* is a string of characters containing at most one function letter and possibly one or more function modifiers. Other arguments to the command are file or directory names specifying the files that are to be restored. Unless the h key is specified (see below), the appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

     The function portion of the key is specified by one of the following letters:

     r   The tape is read and loaded into the current directory. This should not be done lightly; the r key should only be used to restore a complete dump tape onto a clear file system or to restore an incremental dump tape after a full level zero restore. Thus

               /etc/newfs /dev/rrp0g eagle
               /etc/mount /dev/rp0g /mnt
               cd /mnt
               restore r

     is a typical sequence to restore a complete dump. Another *restore* can be done to get an incremental dump in on top of this.
     A *dump*(8) followed by a *newfs*(8) and a *restore* is used to change the size of a file system.

     R   *Restore* requests a particular tape of a multi volume set on which to restart a full restore (see the r key above). This allows *restore* to be interrupted and then restarted.

     x   The named files are extracted from the tape. If the named file matches a directory whose contents had been written onto the tape, and the h key is not specified, the directory is recursively extracted. The owner, modification time, and mode are restored (if possible). If no file argument is given, then the root directory is extracted, which results in the entire content of the tape being extracted, unless the h key has been specified.

     t   The names of the specified files are listed if they occur on the tape. If no file argument is given, then the root directory is listed, which results in the entire content of the tape being listed, unless the h key has been specified. Note that the t key replaces the function of the old *dumpdir* program.

     i   This mode allows interactive restoration of files from a dump tape. After reading in the directory information from the tape, *restore* provides a shell like interface that allows the user to move around the directory tree selecting files to be extracted. The available commands are given below; for those commands that require an argument, the default is the current directory.

          ls [arg] — List the current or specified directory. Entries that are directories are appended with a "/". Entries that have been marked for extraction are prepended with a "*". If the verbose key is set the inode number of each entry is also listed.

          cd arg — Change the current working directory to the specified argument.

          pwd — Print the full pathname of the current working directory.

          add [arg] — The current directory or specified argument is added to the list of files to be

8

extracted. If a directory is specified, then it and all its descendents are added to the extraction list (unless the h key is specified on the command line). Files that are on the extraction list are prepended with a "*" when they are listed by ls.

delete [arg] — The current directory or specified argument is deleted from the list of files to be extracted. If a directory is specified, then it and all its descendents are deleted from the extraction list (unless the h key is specified on the command line). The most expedient way to extract most of the files from a directory is to add the directory to the extraction list and then delete those files that are not needed.

extract — All the files that are on the extraction list are extracted from the dump tape. *Restore* will ask which volume the user wishes to mount. The fastest way to extract a few files is to start with the last volume, and work towards the first volume.

verbose — The sense of the v key is toggled. When set, the verbose key causes the ls command to list the inode numbers of all entries. It also causes *restore* to print out information about each file as it is extracted.

help — List a summary of the available commands.

quit — Restore immediately exits, even if the extraction list is not empty.

The following characters may be used in addition to the letter that selects the function desired.

v   Normally *restore* does its work silently. The v (verbose) key causes it to type the name of each file it treats preceded by its file type.

f   The next argument to *restore* is used as the name of the archive instead of /dev/rmt?. If the name of the file is "−", *restore* reads from standard input. Thus, *dump*(8) and *restore* can be used in a pipeline to dump and restore a file system with the command

            dump 0f - /usr | (cd /mnt; restore xf -)

y   *Restore* will not ask whether it should abort the restore if gets a tape error. It will always try to skip over the bad tape block(s) and continue as best it can.

m   *Restore* will extract by inode numbers rather than by file name. This is useful if only a few files are being extracted, and one wants to avoid regenerating the complete pathname to the file.

h   *Restore* extracts the actual directory, rather than the files that it references. This prevents hierarchical restoration of complete subtrees from the tape.

DIAGNOSTICS

Complaints about bad key characters.

Complaints if it gets a read error. If y has been specified, or the user responds "y", *restore* will attempt to continue the restore.

If the dump extends over more than one tape, *restore* will ask the user to change tapes. If the x or l key has been specified, *restore* will also ask which volume the user wishes to mount. The fastest way to extract a few files is to start with the last volume, and work towards the first volume.

There are numerous consistency checks that can be listed by *restore*. Most checks are self-explanatory or can "never happen". Common errors are given below.

Converting to new file system format.
    A dump tape created from the old file system has been loaded. It is automatically

8

converted to the new file system format.

<filename>: not found on tape
> The specified file name was listed in the tape directory, but was not found on the tape. This is caused by tape read errors while looking for the file, and from using a dump tape created on an active file system.

expected next file <inumber>, got <inumber>.
> A file that was not listed in the directory showed up. This can occur when using a dump tape created on an active file system.

Incremental tape too low
> When doing incremental restore, a tape that was written before the previous incremental tape, or that has too low an incremental level has been loaded.

Incremental tape too high
> When doing incremental restore, a tape that does not begin its coverage where the previous incremental tape left off, or that has too high an incremental level has been loaded.

Tape read error while restoring <filename>
Tape read error while skipping over inode <inumber>
Tape read error while trying to resynchronize
> A tape read error has occurred. If a file name is specified, then its contents are probably partially wrong. If an inode is being skipped or the tape is trying to resynchronize, then no extracted files have been corrupted, though files may not be found on the tape.

resync restore, skipped <num> blocks
> After a tape read error, *restore* may have to resynchronize itself. This message lists the number of blocks that were skipped over.

FILES

/dev/rmt?	the default tape drive
/tmp/rstdir•	file containing directories on the tape.
/tmp/rstmode•	owner, mode, and time stamps for directories.
./restoresymtab	information passed between incremental restores.

SEE ALSO

rrestore(8C) dump(8), newfs(8), mount(8), mkfs(8)

BUGS

Restore can get confused when doing incremental restores from dump tapes that were made on active file systems.

A level zero dump must be done after a full restore. Because restore runs in user code, it has no control over inode allocation; thus a full restore must be done to get a new set of directories reflecting the new inode numbering, even though the contents of the files is unchanged.

8

NAME

rexecd — remote execution server

SYNOPSIS

/etc/rexecd

DESCRIPTION

Rexecd is the server for the *rexec*(3X) routine. The server provides remote execution facilities with authentication based on user names and encrypted passwords.

Rexecd listens for service requests at the port indicated in the "exec" service specification; see *services*(5). When a service request is received the following protocol is initiated:

1) The server reads characters from the socket up to a null ('\0') byte. The resultant string is interpreted as an ASCII number, base 10.

2) If the number received in step 1 is non-zero, it is interpreted as the port number of a secondary stream to be used for the **stderr**. A second connection is then created to the specified port on the client's machine.

3) A null terminated user name of at most 16 characters is retrieved on the initial socket.

4) A null terminated, encrypted, password of at most 16 characters is retrieved on the initial socket.

5) A null terminated command to be passed to a shell is retrieved on the initial socket. The length of the command is limited by the upper bound on the size of the system's argument list.

6) *Rexecd* then validates the user as is done at login time and, if the authentication was successful, changes to the user's home directory, and establishes the user and group protections of the user. If any of these steps fail the connection is aborted with a diagnostic message returned.

7) A null byte is returned on the connection associated with the **stderr** and the command line is passed to the normal login shell of the user. The shell inherits the network connections established by *rexecd*.

DIAGNOSTICS

All diagnostic messages are returned on the connection associated with the **stderr**, after which any network connections are closed. An error is indicated by a leading byte with a value of 1 (0 is returned in step 7 above upon successful completion of all the steps prior to the command execution).

"username too long"
The name is longer than 16 characters.

"password too long"
The password is longer than 16 characters.

"command too long "
The command line passed exceeds the size of the argument list (as configured into the system).

"Login incorrect."
No password file entry for the user name existed.

"Password incorrect."
The wrong was password supplied.

"No remote directory."
The *chdir* command to the home directory failed.

"Try again."
A *fork* by the server failed.

"/bin/sh: ..."
The user's login shell could not be started.

BUGS

Indicating "Login incorrect" as opposed to "Password incorrect" is a security breach which allows people to probe a system for users with null passwords.

A facility to allow all data exchanges to be encrypted should be present.

8

NAME

rlogind — remote login server

SYNOPSIS

/etc/rlogind [−d]

DESCRIPTION

Rlogind is the server for the *rlogin*(1C) program. The server provides a remote login facility with authentication based on privileged port numbers.

Rlogind listens for service requests at the port indicated in the "login" service specification; see *services*(5). When a service request is received the following protocol is initiated:

1) The server checks the client's source port. If the port is not in the range 0-1023, the server aborts the connection.

2) The server checks the client's source address. If the address is associated with a host for which no corresponding entry exists in the host name data base (see *hosts*(5)), the server aborts the connection.

Once the source port and address have been checked, *rlogind* allocates a pseudo terminal (see *pty*(4)), and manipulates file descriptors so that the slave half of the pseudo terminal becomes the **stdin** , **stdout** , and **stderr** for a login process. The login process is an instance of the *login*(1) program, invoked with the −r option. The login process then proceeds with the authentication process as described in *rshd*(8C), but if automatic authentication fails, it reprompts the user to login as one finds on a standard terminal line.

The parent of the login process manipulates the master side of the pseduo terminal, operating as an intermediary between the login process and the client instance of the *rlogin* program. In normal operation, the packet protocol described in *pty*(4) is invoked to provide ^S/^Q type facilities and propagate interrupt signals to the remote programs. The login process propagates the client terminal's baud rate and terminal type, as found in the environment variable, "TERM"; see *environ*(7).

DIAGNOSTICS

All diagnostic messages are returned on the connection associated with the **stderr**, after which any network connections are closed. An error is indicated by a leading byte with a value of 1.

"Hostname for your address unknown."
No entry in the host name database existed for the client's machine.

"Try again."
A *fork* by the server failed.

"/bin/sh: ..."
The user's login shell could not be started.

BUGS

The authentication procedure used here assumes the integrity of each client machine and the connecting medium. This is insecure, but is useful in an "open" environment.

A facility to allow all data exchanges to be encrypted should be present.

8

NAME
     rmt — remote magtape protocol module

SYNOPSIS
     /etc/rmt

DESCRIPTION
     Rmt is a program used by the remote dump and restore programs in manipulating a magnetic tape drive through an interprocess communication connection. *Rmt* is normally started up with an *rexec*(3X) or *rcmd*(3X) call.

     The *rmt* program accepts requests specific to the manipulation of magnetic tapes, performs the commands, then responds with a status indication. All responses are in ASCII and in one of two forms. Successful commands have responses of

          A*number*\n

     where *number* is an ASCII representation of a decimal number. Unsuccessful commands are responded to with

          E*error-number*\n*error-message*\n,

     where *error-number* is one of the possible error numbers described in *intro*(2) and *error-message* is the corresponding error string as printed from a call to *perror*(3). The protocol is comprised of the following commands (a space is present between each token).

     O device mode   Open the specified *device* using the indicated *mode*. *Device* is a full pathname and *mode* is an ASCII representation of a decimal number suitable for passing to *open*(2). If a device had already been opened, it is closed before a new open is performed.

     C device   Close the currently open device. The *device* specified is ignored.

     L whence offset Perform an *lseek*(2) operation using the specified parameters. The response value is that returned from the *lseek* call.

     W count   Write data onto the open device. *Rmt* reads *count* bytes from the connection, aborting if a premature end-of-file is encountered. The response value is that returned from the *write*(2) call.

     R count   Read *count* bytes of data from the open device. If *count* exceeds the size of the data buffer (10 kilobytes), it is truncated to the data buffer size. *Rmt* then performs the requested *read*(2) and responds with A*count-read*\n if the read was successful; otherwise an error in the standard format is returned. If the read was successful, the data read is then sent.

     I operation count

                         Perform a MTIOCOP *ioctl*(2) command using the specified parameters. The parameters are interpreted as the ASCII representations of the decimal values to place in the *mt_op* and *mt_count* fields of the structure used in the *ioctl* call. The return value is the *count* parameter when the operation is successful.

     S   Return the status of the open device, as obtained with a MTIOCGET *ioctl* call. If the operation was successful, an "ack" is sent with the size of the status buffer, then the status buffer is sent (in binary).

     Any other command causes *rmt* to exit.

DIAGNOSTICS
     All responses are of the form described above.

8

SEE ALSO

rcmd(3X), rexec(3X), mtio(4), rdump(8C), rrestore(8C)

BUGS

People tempted to use this for a remote file access protocol are discouraged.

8

NAME

route — manually manipulate the routing tables

SYNOPSIS

/etc/route [−f] [*command args*]

DESCRIPTION

Route is a program used to manually manipulate the network routing tables. It normally is not needed, as the system routing table management daemon, *routed*(8C), should tend to this task.

Route accepts three commands: *add*, to add a route; *delete*, to delete a route; and *change*, to modify an existing route.

All commands have the following syntax:

/etc/route *command* destination gateway [metric]

where *destination* is a host or network for which the route is "to", *gateway* is the gateway to which packets should be addressed, and *metric* is an optional count indicating the number of hops to the *destination*. If no metric is specified, *route* assumes a value of 0. Routes to a particular host are distinguished from those to a network by interpreting the Internet address associated with *destination*. If the *destination* has a "local address part" of INADDR_ANY, then the route is assumed to be to a network; otherwise, it is presumed to be a route to a host. If the route is to a destination connected via a gateway, the *metric* should be greater than 0. All symbolic names specified for a *destination* or *gateway* are looked up first in the host name database, *hosts*(5). If this lookup fails, the name is then looked for in the network name database, *networks*(5).

Route uses a raw socket and the SIOCADDRT and SIOCDELRT *ioctl*'s to do its work. As such, only the super-user may modify the routing tables.

If the −f option is specified, *route* will "flush" the routing tables of all gateway entries. If this is used in conjunction with one of the commands described above, the tables are flushed prior to the command's application.

DIAGNOSTICS

"add %s: gateway %s flags %x"
The specified route is being added to the tables. The values printed are from the routing table entry supplied in the *ioctl* call.

"delete %s: gateway %s flags %x"
As above, but when deleting an entry.

"%s %s done"
When the −f flag is specified, each routing table entry deleted is indicated with a message of this form.

"not in table"
A delete operation was attempted for an entry which wasn't present in the tables.

"routing table overflow"
An add operation was attempted, but the system was low on resources and was unable to allocate memory to create the new entry.

SEE ALSO

intro(4N), routed(8C)

BUGS

The change operation is not implemented, one should add the new route, then delete the old one.

NAME
routed — network routing daemon

SYNOPSIS
/etc/routed [−s] [−q] [−t] [*logfile*]

DESCRIPTION
Routed is invoked at boot time to manage the network routing tables. The routing daemon uses a variant of the Xerox NS Routing Information Protocol in maintaining up to date kernel routing table entries.

In normal operation *routed* listens on *udp*(4P) socket 520 (decimal) for routing information packets. If the host is an internetwork router, it periodically supplies copies of its routing tables to any directly connected hosts and networks.

When *routed* is started, it uses the SIOCGIFCONF *ioctl* to find those directly connected interfaces configured into the system and marked "up" (the software loopback interface is ignored). If multiple interfaces are present, it is assumed the host will forward packets between networks. *Routed* then transmits a *request* packet on each interface (using a broadcast packet if the interface supports it) and enters a loop, listening for *request* and *response* packets from other hosts.

When a *request* packet is received, *routed* formulates a reply based on the information maintained in its internal tables. The *response* packet generated contains a list of known routes, each marked with a "hop count" metric (a count of 16, or greater, is considered "infinite"). The metric associated with each route returned provides a metric *relative to the sender*.

Response packets received by *routed* are used to update the routing tables if one of the following conditions is satisfied:

(1)   No routing table entry exists for the destination network or host, and the metric indicates the destination is "reachable" (i.e. the hop count is not infinite).

(2)   The source host of the packet is the same as the router in the existing routing table entry. That is, updated information is being received from the very internetwork router through which packets for the destination are being routed.

(3)   The existing entry in the routing table has not been updated for some time (defined to be 90 seconds) and the route is at least as cost effective as the current route.

(4)   The new route describes a shorter route to the destination than the one currently stored in the routing tables; the metric of the new route is compared against the one stored in the table to decide this.

When an update is applied, *routed* records the change in its internal tables and generates a *response* packet to all directly connected hosts and networks. *Routed* waits a short period of time (no more than 30 seconds) before modifying the kernel's routing tables to allow possible unstable situations to settle.

In addition to processing incoming packets, *routed* also periodically checks the routing table entries. If an entry has not been updated for 3 minutes, the entry's metric is set to infinity and marked for deletion. Deletions are delayed an additional 60 seconds to insure the invalidation is propagated throughout the internet.

Hosts acting as internetwork routers gratuitously supply their routing tables every 30 seconds to all directly connected hosts and networks.

Supplying the −s option forces *routed* to supply routing information whether it is acting as an internetwork router or not. The −q option is the opposite of the −s option. If the −t option is specified, all packets sent or received are printed on the standard output. In addition, *routed* will not divorce itself from the controlling terminal so that interrupts from the keyboard will kill the process. Any other argument supplied is interpreted as the name of file in which

routed's actions should be logged. This log contains information about any changes to the routing tables and a history of recent messages sent and received which are related to the changed route.

In addition to the facilities described above, *routed* supports the notion of "distant" *passive* and *active* gateways. When *routed* is started up, it reads the file */etc/gateways* to find gateways which may not be identified using the SIOGIFCONF *ioctl*. Gateways specified in this manner should be marked passive if they are not expected to exchange routing information, while gateways marked active should be willing to exchange routing information (i.e. they should have a *routed* process running on the machine). Passive gateways are maintained in the routing tables forever and information regarding their existence is included in any routing information transmitted. Active gateways are treated equally to network interfaces. Routing information is distributed to the gateway and if no routing information is received for a period of the time, the associated route is deleted.

The */etc/gateways* is comprised of a series of lines, each in the following format:

< net | host > *name1* gateway *name2* metric *value* < passive | active >

The net or host keyword indicates if the route is to a network or specific host.

Name1 is the name of the destination network or host. This may be a symbolic name located in */etc/networks* or */etc/hosts*, or an Internet address specified in "dot" notation; see *inet*(3N).

Name2 is the name or address of the gateway to which messages should be forwarded.

Value is a metric indicating the hop count to the destination host or network.

The keyword passive or active indicates if the gateway should be treated as *passive* or *active* (as described above).

FILES
    /etc/gateways   for distant gateways

SEE ALSO
    "Internet Transport Protocols", XSIS 028112, Xerox System Integration Standard.
    udp(4P)

BUGS
    The kernel's routing tables may not correspond to those of *routed* for short periods of time while processes utilizing existing routes exit; the only remedy for this is to place the routing process in the kernel.

    Routed should listen to intelligent interfaces, such as an IMP, and to error protocols, such as ICMP, to gather more information.

8

NAME

rrestore — restore a file system dump across the network

SYNOPSIS

/etc/rrestore [key [name ...]

DESCRIPTION

Rrestore obtains from magnetic tape files saved by a previous *dump*(8). The command is identical in operation to *restore*(8) except the *f* key should be specified and the file supplied should be of the form *machine:device*.

Rrestore creates a remote server, *letc/rmt*, on the client machine to access the tape device.

SEE ALSO

restore(8), rmt(8C)

DIAGNOSTICS

Same as *restore*(8) with a few extra related to the network.

BUGS

8

NAME
    rshd — remote shell server

SYNOPSIS
    /etc/rshd

DESCRIPTION
    Rshd is the server for the *rcmd*(3X) routine and, consequently, for the *rsh*(1C) program. The server provides remote execution facilities with authentication based on privileged port numbers.

    Rshd listens for service requests at the port indicated in the "cmd" service specification; see *services*(5). When a service request is received the following protocol is initiated:

    1)   The server checks the client's source port. If the port is not in the range 0-1023, the server aborts the connection.

    2)   The server reads characters from the socket up to a null ('\0') byte. The resultant string is interpreted as an ASCII number, base 10.

    3)   If the number received in step 1 is non-zero, it is interpreted as the port number of a secondary stream to be used for the stderr. A second connection is then created to the specified port on the client's machine. The source port of this second connection is also in the range 0-1023.

    4)   The server checks the client's source address. If the address is associated with a host for which no corresponding entry exists in the host name data base (see *hosts*(5)), the server aborts the connection.

    5)   A null terminated user name of at most 16 characters is retrieved on the initial socket. This user name is interpreted as a user identity to use on the server's machine.

    6)   A null terminated user name of at most 16 characters is retrieved on the initial socket. This user name is interpreted as the user identity on the client's machine.

    7)   A null terminated command to be passed to a shell is retrieved on the initial socket. The length of the command is limited by the upper bound on the size of the system's argument list.

    8)   *Rshd* then validates the user according to the following steps. The remote user name is looked up in the password file and a *chdir* is performed to the user's home directory. If either the lookup or *chdir* fail, the connection is terminated. If the user is not the super-user, (user id 0), the file */etc/hosts.equiv* is consulted for a list of hosts considered "equivalent". If the client's host name is present in this file, the authentication is considered successful. If the lookup fails, or the user is the super-user, then the file *.rhosts* in the home directory of the remote user is checked for the machine name and identity of the user on the client's machine. If this lookup fails, the connection is terminated.

    9)   A null byte is returned on the connection associated with the stderr and the command line is passed to the normal login shell of the user. The shell inherits the network connections established by *rshd*.

DIAGNOSTICS
    All diagnostic messages are returned on the connection associated with the stderr, after which any network connections are closed. An error is indicated by a leading byte with a value of 1 (0 is returned in step 9 above upon successful completion of all the steps prior to the command execution).

    "locuser too long"
    The name of the user on the client's machine is longer than 16 characters.

"remuser too long"
The name of the user on the remote machine is longer than 16 characters.

"command too long "
The command line passed exceeds the size of the argument list (as configured into the system).

"Hostname for your address unknown."
No entry in the host name database existed for the client's machine.

"Login incorrect."
No password file entry for the user name existed.

"No remote directory."
The *chdir* command to the home directory failed.

"Permission denied."
The authentication procedure described above failed.

"Can't make pipe."
The pipe needed for the **stderr**, wasn't created.

"Try again."
A *fork* by the server failed.

"/bin/sh: ..."
The user's login shell could not be started.

SEE ALSO
rsh(1C), rcmd(3X)

BUGS

The authentication procedure used here assumes the integrity of each client machine and the connecting medium. This is insecure, but is useful in an "open" environment.

A facility to allow all data exchanges to be encrypted should be present.

8

NAME
     rwhod — system status server

SYNOPSIS
     /etc/rwhod

DESCRIPTION
     Rwhod is the server which maintains the database used by the *rwho*(1C) and *ruptime*(1C) pro-
     grams. Its operation is predicated on the ability to *broadcast* messages on a network.

     Rwhod operates as both a producer and consumer of status information. As a producer of
     information it periodically queries the state of. the system and constructs status messages which
     are broadcast on a network. As a consumer of information, it listens for other *rwhod* servers'
     status messages, validating them, then recording them in a collection of files located in the
     directory */usr/spool/rwho*.

     The *rwho* server transmits and receives messages at the port indicated in the "rwho" service
     specification, see *services*(5). The messages sent and received, are of the form:

     struct   outmp {
             char   out_line[8];/• tty name •/
             char   out_name[8];/• user id •/
             long   out_time;/• time on •/
     };

     struct   whod {
             char   wd_vers;
             char   wd_type;
             char   wd_fill[2];
             int   wd_sendtime;
             int   wd_recvtime;
             char   wd_hostname[32];
             int   wd_loadav[3];
             int   wd_boottime;
             struct   whoent {
                 struct   outmp we_utmp;
                 int   we_idle;
             } wd_we[1024 / sizeof (struct whoent)];
     };

     All fields are converted to network byte order prior to transmission. The load averages are as
     calculated by the *w*(1) program, and represent load averages over the 5, 10, and 15 minute
     intervals prior to a server's transmission. The host name included is that returned by the
     gethostname(2) system call. The array at the end of the message contains information about the
     users logged in to the sending machine. This information includes the contents of the *utmp*(5)
     entry for each non-idle terminal line and a value indicating the time since a character was last
     received on the terminal line.

     Messages received by the *rwho* server are discarded unless they originated at a *rwho* server's
     port. In addition, if the host's name, as specified in the message, contains any unprintable
     ASCII characters, the message is discarded. Valid messages received by *rwhod* are placed in
     files named *whod.hostname* in the directory */usr/spool/rwho*. These files contain only the most
     recent message, in the format described above.

     Status messages are generated approximately once every 60 seconds. *Rwhod* performs an
     nlist(3) on /vmunix every 10 minutes to guard against the possibility that this file is not the
     system image currently operating.

8

SEE ALSO
>   rwho(1C), ruptime(1C)

BUGS
>   Should relay status information between networks. People often interpret the server dieing as a
>   machine going down.

NAME
     rxformat — format floppy disks

SYNOPSIS
     /etc/rxformat [−d] special

DESCRIPTION
     The *rxformat* program formats a diskette in the specified drive associated with the special device
     special. (*Special* is normally /dev/rx0, for drive 0, or /dev/rx1, for drive 1.) By default, the
     diskette is formatted single density; a −d flag may be supplied to force double density format-
     ting. Single density is compatible with the IBM 3740 standard (128 bytes/sector). In double
     density, each sector contains 256 bytes of data.

     Before formatting a diskette *rxformat* prompts for verification (this allows a user to cleanly abort
     the operation; note that formatting a diskette will destroy any existing data). Formatting is
     done by the hardware. All sectors are zero-filled.

DIAGNOSTICS
     'No such device' means that the drive is not ready, usually because no disk is in the drive or
     the drive door is open. Other error messages are selfexplanatory.

FILES
     /dev/rx?

SEE ALSO
     rx(4V)

BUGS
     A floppy may not be formatted if the header info on sector 1, track 0 has been damaged.
     Hence, it is not possible to format a completely degaussed disk. (This is actually a problem in
     the hardware.)

AUTHOR
     Helge Skrivervik

NAME

sa, accton — system accounting

SYNOPSIS

/etc/sa [**−abcdDfijkKlnrstuv**] [file]

/etc/accton [file]

DESCRIPTION

With an argument naming an existing *file, accton* causes system accounting information for every process executed to be placed at the end of the file. If no argument is given, accounting is turned off.

Sa reports on, cleans up, and generally maintains accounting files.

Sa is able to condense the information in */usr/adm/acct* into a summary file */usr/adm/savacct* which contains a count of the number of times each command was called and the time resources consumed. This condensation is desirable because on a large system */usr/adm/acct* can grow by 100 blocks per day. The summary file is normally read before the accounting file, so the reports include all available information.

If a file name is given as the last argument, that file will be treated as the accounting file; */usr/adm/acct* is the default.

Output fields are labeled: "cpu" for the sum of user+system time (in minutes), "re" for real time (also in minutes), "k" for cpu-time averaged core usage (in 1k units), "avio" for average number of i/o operations per execution. With options fields labeled "tio" for total i/o operations, "k∙sec" for cpu storage integral (kilo-core seconds), "u" and "s" for user and system cpu time alone (both in minutes) will sometimes appear.

There are near a googol of options:

a   Place all command names containing unprintable characters and those used only once under the name '∙∙∙other.'

b   Sort output by sum of user and system time divided by number of calls. Default sort is by sum of user and system times.

c   Besides total user, system, and real time for each command print percentage of total time over all commands.

d   Sort by average number of disk i/o operations.

D   Print and sort by total number of disk i/o operations.

f   Force no interactive threshold compression with −v flag.

i   Don't read in summary file.

j   Instead of total minutes time for each category, give seconds per call.

k   Sort by cpu-time average memory usage.

K   Print and sort by cpu-storage integral.

l   Separate system and user time; normally they are combined.

m   Print number of processes and number of CPU minutes for each user.

n   Sort by number of calls.

r   Reverse order of sort.

s   Merge accounting file into summary file */usr/adm/savacct* when done.

t   For each command report ratio of real time to the sum of user and system times.

u   Superseding all other flags, print for each command in the accounting file the user ID

8

and command name.

v   Followed by a number *n*, types the name of each command used *n* times or fewer.
           Await a reply from the terminal; if it begins with 'y', add the command to the category
           '••junk••.' This is used to strip out garbage.

FILES

/usr/adm/acct   raw accounting
/usr/adm/savacct   summary
/usr/adm/usracct   per-user summary

SEE ALSO

ac(8), acct(2)

BUGS

The number of options to this program is absurd.

NAME

　　savecore — save a core dump of the operating system

SYNOPSIS

　　/etc/savecore *dirname* [*system*]

DESCRIPTION

　　Savecore is meant to be called near the end of the /etc/rc file. Its function is to save the core dump of the system (assuming one was made) and to write a reboot message in the shutdown log.

　　Savecore checks the core dump to be certain it corresponds with the current running unix. If it does it saves the core image in the file *dirname*/vmcore.n and it's brother, the namelist, *dirname*/vmunix.n The trailing ".n" in the pathnames is replaced by a number which grows every time *savecore* is run in that directory.

　　Before savecore writes out a core image, it reads a number from the file *dirname*/minfree. If there are fewer free blocks on the filesystem which contains *dirname* than the number obtained from the minfree file, the core dump is not done. If the minfree file does not exist, savecore always writes out the core file (assuming that a core dump was taken).

　　Savecore also writes a reboot message in the shut down log. If the system crashed as a result of a panic, *savecore* records the panic string in the shut down log too.

　　If the core dump was from a system other than /vmunix, the name of that system must be supplied as *sysname*.

FILES

　　/usr/adm/shutdownlog　　shut down log
　　/vmunix　　　　　　　　　current UNIX

BUGS

　　Can be fooled into thinking a core dump is the wrong size.

8

NAME

sendmail − send mail over the internet

SYNOPSIS

/usr/lib/sendmail [flags] [address ...]

newaliases

mailq

DESCRIPTION

Sendmail sends a message to one or more people, routing the message over whatever networks are necessary. *Sendmail* does internetwork forwarding as necessary to deliver the message to the correct place.

Sendmail is not intended as a user interface routine; other programs provide user-friendly front ends; *sendmail* is used only to deliver pre-formatted messages.

With no flags, *sendmail* reads its standard input up to a control-D or a line with a single dot and sends a copy of the letter found there to all of the addresses listed. It determines the network to use based on the syntax and contents of the addresses.

Local addresses are looked up in a file and aliased appropriately. Aliasing can be prevented by preceding the address with a backslash. Normally the sender is not included in any alias expansions, e.g., if 'john' sends to 'group', and 'group' includes 'john' in the expansion, then the letter will not be delivered to 'john'.

Flags are:

−ba	Go into ARPANET mode. All input lines must end with a CR-LF, and all messages will be generated with a CR-LF at the end. Also, the "From:" and "Sender:" fields are examined for the name of the sender.
−bd	Run as a daemon. This requires Berkeley IPC.
−bi	Initialize the alias database.
−bm	Deliver mail in the usual way (default).
−bp	Print a listing of the queue.
−bs	Use the SMTP protocol as described in RFC821. This flag implies all the operations of the −ba flag that are compatible with SMTP.
−bt	Run in address test mode. This mode reads addresses and shows the steps in parsing; it is used for debugging configuration tables.
−bv	Verify names only − do not try to collect or deliver a message. Verify mode is normally used for validating users or mailing lists.
−bz	Create the configuration freeze file.
−C*file*	Use alternate configuration file.
−d*X*	Set debugging value to *X*.
−F*fullname*	Set the full name of the sender.
−f*name*	Sets the name of the "from" person (i.e., the sender of the mail). −f can only be used by the special users *root, daemon,* and *network,* or if the person you are trying to become is the same as the person you are.
−h*N*	Set the hop count to *N*. The hop count is incremented every time the mail is processed. When it reaches a limit, the mail is returned with an error message, the victim of an aliasing loop.
−n	Don't do aliasing.

8

—o*x value*	Set option *x* to the specified *value*. Options are described below.
—q[*time*]	Processed saved messages in the queue at given intervals. If is omitted, process the queue once. is given as a tagged number, with 's' being seconds, 'm' being minutes, 'h' being hours, 'd' being days, and 'w' being weeks. For example, "—q1h30m" or "—q90m" would both set the timeout to one hour thirty minutes.
—r*name*	An alternate and obsolete form of the —f flag.
—t	Read message for recipients. To:, Cc:, and Bcc: lines will be scanned for people to send to. The Bcc: line will be deleted before transmission. Any addresses in the argument list will be suppressed.
—v	Go into verbose mode. Alias expansions will be announced, etc.

There are also a number of processing options that may be set. Normally these will only be used by a system administrator. Options may be set either on the command line using the —o flag or in the configuration file. These are described in detail in the *Installation and Operation Guide*. The options are:

A*file*	Use alternate alias file.
c	On mailers that are considered "expensive" to connect to, don't initiate immediate connection. This requires queueing.
d*x*	Set the delivery mode to *x*. Delivery modes are 'i' for interactive (synchronous) delivery, 'b' for background (asynchronous) delivery, and 'q' for queue only — i.e., actual delivery is done the next time the queue is run.
D	Try to automatically rebuild the alias database if necessary.
e*x*	Set error processing to mode *x*. Valid modes are 'm' to mail back the error message, 'w' to "write" back the error message (or mail it back if the sender is not logged in), 'p' to print the errors on the terminal (default), 'q' to throw away error messages (only exit status is returned), and 'e' to do special processing for the BerkNet. If the text of the message is not mailed back by modes 'm' or 'w' and if the sender is local to this machine, a copy of the message is appended to the file "dead.letter" in the sender's home directory.
F*mode*	The mode to use when creating temporary files.
f	Save UNIX-style From lines at the front of messages.
g*N*	The default group id to use when calling mailers.
H*file*	The SMTP help file.
i	Do not take dots on a line by themselves as a message terminator.
L*n*	The log level.
m	Send to "me" (the sender) also if I am in an alias expansion.
o	If set, this message may have old style headers. If not set, this message is guaranteed to have new style headers (i.e., commas instead of spaces between addresses). If set, an adaptive algorithm is used that will correctly determine the header format in most cases.
Q*queuedir*	Select the directory in which to queue messages.
r*timeout*	The timeout on reads; if none is set, *sendmail* will wait forever for a mailer.
S*file*	Save statistics in the named file.

8

s   Always instantiate the queue file, even under circumstances where it is not
                     strictly necessary.

T *time*   Set the timeout on messages in the queue to the specified time. After sit-
                     ting in the queue for this amount of time, they will be returned to the
                     sender. The default is three days.

t *stz, dtz*   Set the name of the time zone.

u *N*   Set the default user id for mailers.

If the first character of the user name is a vertical bar, the rest of the user name is used as the
name of a program to pipe the mail to. It may be necessary to quote the name of the user to
keep *sendmail* from suppressing the blanks from between arguments.

Sendmail returns an exit status describing what it did. The codes are defined in < *sysexits.h* >

EX_OK	Successful completion on all addresses.
EX_NOUSER	User name not recognized.
EX_UNAVAILABLE	Catchall meaning necessary resources were not available.
EX_SYNTAX	Syntax error in address.
EX_SOFTWARE	Internal software error, including bad arguments.
EX_OSERR	Temporary operating system error, such as "cannot fork".
EX_NOHOST	Host name not recognized.
EX_TEMPFAIL	Message could not be sent immediately, but was queued.

If invoked as *newaliases, sendmail* will rebuild the alias database. If invoked as *mailq, sendmail*
will print the contents of the mail queue.

FILES

Except for /usr/lib/sendmail.cf, these pathnames are all specified in /usr/lib/sendmail.cf.
Thus, these values are only approximations.

/usr/lib/aliases	raw data for alias names
/usr/lib/aliases.pag	
/usr/lib/aliases.dir	data base of alias names
/usr/lib/sendmail.cf	configuration file
/usr/lib/sendmail.fc	frozen configuration
/usr/lib/sendmail.hf	help file
/usr/lib/sendmail.st	collected statistics
/usr/bin/uux	to deliver uucp mail
/usr/net/bin/v6mail	to deliver local mail
/usr/net/bin/sendberkmail	to deliver Berknet mail
/usr/lib/mailers/arpa	to deliver ARPANET mail
/usr/spool/mqueue/∗	temp files

SEE ALSO

biff(1), binmail(1), mail(1), aliases(5), sendmail.cf(5), rmail(1), mailaddr(7);
DARPA Internet Request For Comments RFC819, RFC821, RFC822;
Sendmail — An Internetwork Mail Router;
Sendmail Installation and Operation Guide.

BUGS

Sendmail converts blanks in addresses to dots. This is incorrect according to the old ARPANET
mail protocol RFC733 (NIC 41952), but is consistent with the new protocols (RFC822).

8

NAME
　　　　shutdown — close down the system at a given time

SYNOPSIS
　　　　/etc/shutdown [−k] [−r] [−h] time [warning-message ...]

DESCRIPTION
　　　　Shutdown provides an automated shutdown procedure which a super-user can use to notify
　　　　users nicely when the system is shutting down, saving them from system administrators, hack-
　　　　ers, and gurus, who would otherwise not bother with niceties.

　　　　Time is the time at which *shutdown* will bring the system down and may be the word now (indi-
　　　　cating an immediate shutdown) or specify a future time in one of two formats: +number and
　　　　hour:min. The first form brings the system down in *number* minutes and the second brings the
　　　　system down at the time of day indicated (as a 24−hour clock).

　　　　At intervals which get closer together as apocalypse approaches, warning messages are displayed
　　　　at the terminals of all users on the system. Five minutes before shutdown, or immediately if
　　　　shutdown is in less than 5 minutes, logins are disabled by creating /etc/nologin and writing a
　　　　message there. If this file exists when a user attempts to log in, *login*(1) prints its contents and
　　　　exits. The file is removed just before *shutdown* exits.

　　　　At shutdown time a message is written in the file /usr/adm/shutdownlog, containing the time
　　　　of shutdown, who ran shutdown and the reason. Then a terminate signal is sent at *init* to bring
　　　　the system down to single-user state. Alternatively, if −r, −h, or −k was used, then *shutdown*
　　　　will exec *reboot*(8), *halt*(8), or avoid shutting the system down (respectively). (If it isn't obvi-
　　　　ous, −k is to make people *think* the system is going down!)

　　　　The time of the shutdown and the warning message are placed in /etc/nologin and should be
　　　　used to inform the users about when the system will be back up and why it is going down (or
　　　　anything else).

FILES
　　　　/etc/nologin　　　tells login not to let anyone log in
　　　　/usr/adm/shutdownlog log file for successful shutdowns.

SEE ALSO
　　　　login(1), reboot(8)

BUGS
　　　　Only allows you to kill the system between now and 23:59 if you use the absolute time for
　　　　shutdown.

8

NAME

    sticky — executable files with persistent text

DESCRIPTION

    While the 'sticky bit', mode 01000 (see *chmod*(2)), is set on a sharable executable file, the text of that file will not be removed from the system swap area. Thus the file does not have to be fetched from the file system upon each execution. As long as a copy remains in the swap area, the original text cannot be overwritten in the file system, nor can the file be deleted. (Directory entries can be removed so long as one link remains.)

    Sharable files are made by the **—n** and **—z** options of *ld*(1).

    To replace a sticky file that has been used do: (1) Clear the sticky bit with *chmod*(1). (2) Execute the old program to flush the swapped copy. This can be done safely even if others are using it. (3) Overwrite the sticky file. If the file is being executed by any process, writing will be prevented; it suffices to simply remove the file and then rewrite it, being careful to reset the owner and mode with *chmod* and *chown*(2). (4) Set the sticky bit again.

    Only the super-user can set the sticky bit.

BUGS

    Are self-evident.

    Is largely unnecessary on the VAX; matters only for large programs that will page heavily to start, since text pages are normally cached incore as long as possible after all instances of a text image exit.

NAME
    swapon — specify additional device for paging and swapping

SYNOPSIS
    /etc/swapon —a
    /etc/swapon name ...

DESCRIPTION
    Swapon is used to specify additional devices on which paging and swapping are to take place.
    The system begins by swapping and paging on only a single device so that only one disk is
    required at bootstrap time. Calls to *swapon* normally occur in the system multi-user initializa-
    tion file */etc/rc* making all swap devices available, so that the paging and swapping activity is
    interleaved across several devices.

    Normally, the —a argument is given, causing all devices marked as "sw" swap devices in
    /etc/fstab to be made available.

    The second form gives individual block devices as given in the system swap configuration table.
    The call makes only this space available to the system for swap allocation.

SEE ALSO
    swapon(2), init(8)

FILES
    /dev/[ru][pk]?b   normal paging devices

BUGS
    There is no way to stop paging and swapping on a device. It is therefore not possible to make
    use of devices which may be dismounted during system operation.

8

NAME

sync — update the super block

SYNOPSIS

/etc/sync

DESCRIPTION

Sync executes the *sync* system primitive. *Sync* can be called to insure all disk writes have been completed before the processor is halted in a way not suitably done by *reboot*(8) or *halt*(8).

See *sync*(2) for details on the system primitive.

SEE ALSO

sync(2), fsync(2), halt(8), reboot(8), update(8)

NAME

syslog — log systems messages

SYNOPSIS

/etc/syslog [−m*N*] [−f*name*] [−d]

DESCRIPTION

Syslog reads a datagram socket and logs each line it reads into a set of files described by the configuration file /etc/syslog.conf. *Syslog* configures when it starts up and whenever it receives a hangup signal.

Each message is one line. A message can contain a priority code, marked by a digit in angle braces at the beginning of the line. Priorities are defined in <*syslog.h*>, as follows:

LOG_ALERT   this priority should essentially never be used. It applies only to messages that are so important that every user should be aware of them, e.g., a serious hardware failure.

LOG_SALERT   messages of this priority should be issued only when immediate attention is needed by a qualified system person, e.g., when some valuable system resource dissappears. They get sent to a list of system people.

LOG_EMERG   Emergency messages are not sent to users, but represent major conditions. An example might be hard disk failures. These could be logged in a separate file so that critical conditions could be easily scanned.

LOG_ERR   these represent error conditions, such as soft disk failures, etc.

LOG_CRIT   such messages contain critical information, but which can not be classed as errors, for example, 'su' attempts. Messages of this priority and higher are typically logged on the system console.

LOG_WARNING   issued when an abnormal condition has been detected, but recovery can take place.

LOG_NOTICE   something that falls in the class of "important information"; this class is informational but important enough that you don't want to throw it away casually. Messages without any priority assigned to them are typically mapped into this priority.

LOG_INFO   information level messages. These messages could be thrown away without problems, but should be included if you want to keep a close watch on your system.

LOG_DEBUG   it may be useful to log certain debugging information. Normally this will be thrown away.

It is expected that the kernel will not log anything below LOG_ERR priority.

The configuration file is in two sections separated by a blank line. The first section defines files that *syslog* will log into. Each line contains a single digit which defines the lowest priority (highest numbered priority) that this file will receive, an optional asterisk which guarantees that something gets output at least every 20 minutes, and a pathname. The second part of the file contains a list of users that will be informed on SALERT level messages. For example, the configuration file:

```
5*/dev/console
8/usr/spool/adm/syslog
3/usr/adm/critical

eric
```

8

        kridle
        kalash

logs all messages of priority 5 or higher onto the system console, including timing marks every 20 minutes; all messages of priority 8 or higher into the file /usr/spool/adm/syslog; and all messages of priority 3 or higher into /usr/adm/critical. The users "eric", "kridle", and "kalash" will be informed on any subalert messages.

The flags are:

−m   Set the mark interval to *N* (default 20 minutes).

−f   Specify an alternate configuration file.

−d   Turn on debugging (if compiled in).

To bring *syslog* down, it should be sent a terminate signal. It logs that it is going down and then waits approximately 30 seconds for any additional messages to come in.

There are some special messages that cause control functions. "<∗>N" sets the default message priority to *N*. "<$>" causes *syslog* to reconfigure (equivalent to a hangup signal). This can be used in a shell file run automatically early in the morning to truncate the log.

Syslog creates the file /etc/syslog.pid if possible containing a single line with its process id. This can be used to kill or reconfigure *syslog*.

FILES

    /etc/syslog.conf − the configuration file
    /etc/syslog.pid − the process id

BUGS

    LOG_ALERT and LOG_SUBALERT messages should only be allowed to privileged programs.

    Actually, *syslog* is not clever enough to deal with kernel error messages in the current implementation.

SEE ALSO

    syslog(3)

8

NAME

  telnetd — DARPA TELNET protocol server

SYNOPSIS

  /etc/telnetd [−d] [*port*]

DESCRIPTION

  Telnetd is a server which supports the DARPA standard TELNET virtual terminal protocol. The TELNET server operates at the port indicated in the "telnet" service description; see *services*(5). This port number may be overridden (for debugging purposes) by specifying a port number on the command line. If the −d option is specified, each socket created by *telnetd* will have debugging enabled (see SO_DEBUG in *socket*(2)).

  Telnetd operates by allocating a pseudo-terminal device (see *pty*(4)) for a client, then creating a login process which has the slave side of the pseudo-terminal as **stdin**, **stdout**, and **stderr**. *Telnetd* manipulates the master side of the pseudo terminal, implementing the TELNET protocol and passing characters between the client and login process.

  When a TELNET session is started up, *telnetd* sends a TELNET option to the client side indicating a willingness to do "remote echo" of characters. The pseudo terminal allocated to the client is configured to operate in "cooked" mode, and with XTABS and CRMOD enabled (see *tty*(4)). Aside from this initial setup, the only mode changes *telnetd* will carry out are those required for echoing characters at the client side of the connection.

  Telnetd supports binary mode, and most of the common TELNET options, but does not, for instance, support timing marks. Consult the source code for an exact list of which options are not implemented.

SEE ALSO

  telnet(1C)

BUGS

  A complete list of the options supported should be given here.

NAME

tftpd — DARPA Trivial File Transfer Protocol server

SYNOPSIS

/etc/tftpd [−d] [*port*]

DESCRIPTION

Tftpd is a server which supports the DARPA Trivial File Transfer Protocol. The TFTP server operates at the port indicated in the "tftp" service description; see *services*(5). This port number may be overridden (for debugging purposes) by specifying a port number on the command line. If the −d option is specified, each socket created by *tftpd* will have debugging enabled (see SO_DEBUG in *socket*(2)).

The use of *tftp* does not require an account or password on the remote system. Due to the lack of authentication information, *tftpd* will allow only publicly readable files to be accessed. Note that this extends the concept of "public" to include all users on all hosts that can be reached through the network; this may not be appropriate on all systems, and its implications should be considered before enabling tftp service.

SEE ALSO

tftp(1C)

BUGS

This server is known only to be self consistent (i.e. it operates with the user TFTP program, *tftp*(1C)). Due to the unreliability of the transport protocol (UDP) and the scarcity of TFTP implementations, it is uncertain whether it really works.

The search permissions of the directories leading to the files accessed are not checked.

8

NAME

　　trpt — transliterate protocol trace

SYNOPSIS

　　trpt [−a] [−s] [−t] [−J] [−p hex-address] [system [core]]

DESCRIPTION

　　Trpt interrogates the buffer of TCP trace records created when a socket is marked for "debugging" (see *setsockopt*(2)), and prints a readable description of these records. When no options are supplied, *trpt* prints all the trace records found in the system grouped according to TCP connection protocol control block (PCB). The following options may be used to alter this behavior.

　　−s　　in addition to the normal output, print a detailed description of the packet sequencing information,

　　−t　　in addition to the normal output, print the values for all timers at each point in the trace,

　　−J　　just give a list of the protocol control block addresses for which there are trace records,

　　−p　　show only trace records associated with the protocol control block who's address follows,

　　−a　　in addition to the normal output, print the values of the source and destination addresses for each packet recorded.

　　The recommended use of *trpt* is as follows. Isolate the problem and enable debugging on the socket(s) involved in the connection. Find the address of the protocol control blocks associated with the sockets using the −A option to *netstat*(1). Then run *trpt* with the −p option, supplying the associated protocol control block addresses. If there are many sockets using the debugging option, the −J option may be useful in checking to see if any trace records are present for the socket in question.

　　If debugging is being performed on a system or core file other than the default, the last two arguments may be used to supplant the defaults.

FILES

　　/vmunix
　　/dev/kmem

SEE ALSO

　　setsockopt(2), netstat(1)

DIAGNOSTICS

　　"no namelist" when the system image doesn't contain the proper symbols to find the trace buffer; others which should be self explanatory.

BUGS

　　Should also print the data for each input or output, but this is not saved in the race record.

　　The output format is inscrutable and should be described here.

8

NAME
     tunefs — tune up an existing file system

SYNOPSIS
     /etc/tunefs *tuneup-options special\filesys*

DESCRIPTION
     Tunefs is designed to change the dynamic parameters of a file system which affect the layout policies. The parameters which are to be changed are indicated by the flags given below:

     −a maxcontig
          This specifies the maximum number of contiguous blocks that will be laid out before forcing a rotational delay (see −d below). The default value is one, since most device drivers require an interrupt per disk transfer. Device drivers that can chain several buffers together in a single transfer should set this to the maximum chain length.

     −d rotdelay
          This specifies the expected time (in milliseconds) to service a transfer completion interrupt and initiate a new transfer on the same disk. It is used to decide how much rotational spacing to place between successive blocks in a file.

     −e maxbpg
          This indicates the maximum number of blocks any single file can allocate out of a cylinder group before it is forced to begin allocating blocks from another cylinder group. Typically this value is set to about one quarter of the total blocks in a cylinder group. The intent is to prevent any single file from using up all the blocks in a single cylinder group, thus degrading access times for all files subsequently allocated in that cylinder group. The effect of this limit is to cause big files to do long seeks more frequently than if they were allowed to allocate all the blocks in a cylinder group before seeking elsewhere. For file systems with exclusively large files, this parameter should be set higher.

     −m minfree
          This value specifies the percentage of space held back from normal users; the minimum free space threshold. The default value used is 10%. This value can be set to zero, however up to a factor of three in throughput will be lost over the performance obtained at a 10% threshold. Note that if the value is raised above the current usage level, users will be unable to allocate files until enough files have been deleted to get under the higher threshold.

SEE ALSO
     fs(5), newfs(8), mkfs(8)

     McKusick, Joy, Leffler; "A Fast File System for Unix", Computer Systems Research Group, Dept of EECS, Berkeley, CA 94720; TR #7, September 1982.

BUGS
     This program should work on mounted and active file systems. Because the super-block is not kept in the buffer cache, the program will only take effect if it is run on dismounted file systems. (if run on the root file system, the system must be rebooted)

     You can tune a file system, but you can't tune a fish.

8

NAME

update — periodically update the super block

SYNOPSIS

/etc/update

DESCRIPTION

Update is a program that executes the *sync*(2) primitive every 30 seconds. This insures that the file system is fairly up to date in case of a crash. This command should not be executed directly, but should be executed out of the initialization shell command file.

SEE ALSO

sync(2), sync(8), init(8), rc(8)

BUGS

With *update* running, if the CPU is halted just as the *sync* is executed, a file system can be damaged. This is partially due to DEC hardware that writes zeros when NPR requests fail. A fix would be to have *sync*(8) temporarily increment the system time by at least 30 seconds to trigger the execution of *update*. This would give 30 seconds grace to halt the CPU.

8

NAME

     uuclean — uucp spool directory clean-up

SYNOPSIS

     uuclean [option] ...

DESCRIPTION

     Uuclean will scan the spool directory for files with the specified prefix and delete all those which
     are older than the specified number of hours.

     The following options are available.

     —p*pre*   Scan for files with *pre* as the file prefix. Up to 10 **—p** arguments may be specified. A
               —p without any *pre* following will cause all files older than the specified time to be
               deleted.

     —n*time*   Files whose age is more than *time* hours will be deleted if the prefix test is satisfied.
               (default time is 72 hours)

     —m   Send mail to the owner of the file when it is deleted.

     This program will typically be started by *cron*(8).

FILES

     /usr/lib/uucp   directory with commands used by uuclean internally

     /usr/lib/uucp/spool   spool directory

SEE ALSO

     uucp(1C), uux(1C)

8

NAME

 uusnap — show snapshot of the UUCP system

SYNOPSIS

 uusnap

DESCRIPTION

 Uusnap displays in tabular format a synopsis of the current UUCP situation. The format of each line is as follows:

 site N Cmds N Data N Xqts Message

 Where "site" is the name of the site with work, "N" is a count of each of the three possible types of work (command, data, or remote execute), and "Message" is the current status message for that site as found in the STST file.

 Included in "Message" may be the time left before UUCP can re-try the call, and the count of the number of times that UUCP has tried to reach the site.

SEE ALSO

 uucp(1C), UUCP Implementation Guide

8

NAME

vipw — edit the password file

SYNOPSIS

vipw

DESCRIPTION

Vipw edits the password file while setting the appropriate locks, and does any necessary processing after the password file is unlocked. If the password file is already being edited, then you will be told to try again later. The *vi* editor will be used unless the environment variable EDITOR indicates an alternate editor. *Vipw* performs a number of consistency checks on the password entry for *root*, and will not allow a password file with a "mangled" root entry to be installed.

SEE ALSO

chfn(1), chsh(1), passwd(1), passwd(5), adduser(8)

FILES

/etc/ptmp

Installing and Operating 4.2BSD on the VAX
July 21, 1983

Samuel J. Leffler

William N. Joy

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720
(415) 642-7780

ABSTRACT

This document contains instructions for the installation and operation of the 4.2BSD release of the VAX* UNIX** system, as distributed by U. C. Berkeley.

It discusses procedures for installing UNIX on a new VAX, and for upgrading an existing VAX UNIX system to the new release. An explanation of how to how to lay out file systems on available disks, how to set up terminal lines and user accounts, and how to perform system-specific tailoring is provided. A description of how to install and configure the networking facilities included with 4.2BSD is included. Finally, the document details system operation procedures— shutdown and startup, hardware error reporting and diagnosis, file system backup procedures, resource control, performance monitoring, and procedures for recompiling and reinstalling system software.

---

* DEC, VAX, IDC, UNIBUS and MASSBUS are trademarks of Digital Equipment Corporation.
** UNIX is a Trademark of Bell Laboratories.

September 22, 1983

1. INTRODUCTION

This document explains how to install the 4.2BSD release of the Berkeley version of UNIX for the VAX on your system. Due to the new file system organization used in 4.2BSD, no matter what version of UNIX you may currently be running you will have to perform a full bootstrap from the distribution tape; the techniques for converting "old" systems are discussed in a chapter 3 of this document.

1.1. Hardware supported

This distribution can be booted on a VAX 11/780, VAX 11/750, or VAX 11/730 cpu with any of the following disks:

DEC MASSBUS:	RM03, RM05, RM80, RP06, RP07
EMULEX MASSBUS:	AMPEX 300M, 330M, CDC 300M, FUJITSU 404M
DEC UNIBUS:	RK07, RA80, RA81, RA60
EMULEX SC-21V UNIBUS*:	AMPEX 300M, 330M, CDC 300M, FUJITSU 160M, 404M
DEC IDC:	R80, RL02

The tape drives supported by this distribution are:

DEC MASSBUS:	TE16, TU45, TU77, TU78
DEC UNIBUS:	TS11, TU80
EMULEX TC-11 UNIBUS:	KENNEDY 9300, CIPHER
TU45 UNIBUS*:	SI 9700

The tapes and disks may be on any available UNIBUS or MASSBUS adapter at any slot with the proviso that the tape device must be slave number 0 on the formatter if it is a MASSBUS tape drive.

1.2. Distribution format

The basic distribution contains the following items:

(2)   1600bpi 2400' magnetic tapes,
(1)   TU58 console cassette, and
(1)   RX01 console floppy disk.

Installation on any machine requires a tape unit. Since certain standard VAX packages do not include a tape drive, this means one must either borrow one from another VAX system or one must be purchased separately. The console media distributed with the system are not suitable for use as the standard console media; their intended use is only for installation.

The distribution does not fit on several standard VAX configurations which contain only small disks. If your hardware configuration does not provide at least 75 Megabytes of disk space you can still install the distribution, but you will probably have to operate without source for the user level commands and, possibly, the source for the operating system. The previous RK07-only distribution format provided by our group is no longer available. Further, no attempt has ever been made to install the system on the standard VAX-11/730 hardware configuration from DEC which contains only dual RL02 disk drives (though the distribution tape may be bootstrapped on an RL11 controller and the system provides support for RL02 disk

---

* Other UNIBUS controllers and drives may be easily usable with the system, but will likely require minor modifications to the system to allow bootstrapping. The EMULEX disk and SI tape controllers, and the drives shown here are known to work as bootstrap devices.

drives either on an IDC or an RL11). The labels on the two distribution tapes indicate the amount of disk space each tape file occupies, these should be used in selecting file system layouts on systems with little disk space.

If you have the facilities, it is a good idea immediately to copy the magnetic tapes in the distribution kit to guard against disaster. The tapes are 9-track 1600 BPI and contain some 512-byte records followed by many 10240-byte records. There are interspersed tape marks; end-of-tape is signaled by a double end-of-file.

The basic bootstrap material is present in three short files at the beginning of the bootstrap tape. The first file on the tape contains preliminary bootstrapping programs. This is followed by a binary image of a 400 kilobyte "mini root" file system. Following the mini root file is a full dump of the root file system (see *dump* (8)**). Additional files on the first and second tapes contain tape archive images (see *tar* (1)): the fourth file on the first tape contains source for the system (/sys); the fifth file on the first tape contains most of the files in the file system /usr, except the source (/usr/src) which is in the first file on the second tape. The second file on the second tape contains software contributed by the user community, refer to the accompanying documentation for a description of its contents and an explanation of how it should be installed.

1.3. VAX hardware terminology

This section gives a short discussion of VAX hardware terminology to help you get your bearings.

If you have MASSBUS disks and tapes it is necessary to know the MASSBUS they are attached to, at least for the purposes of bootstrapping and system description. The MASSBUSes can have up to 8 devices attached to them. A disk counts as a device. A tape *formatter* counts as a device, and several tape drives may be attached to a formatter. If you have a separate MASSBUS adapter for a disk and one for a tape then it is conventional to put the disk as unit 0 on the MASSBUS with the lowest "TR" number, and the tape formatter as unit 0 on the next MASSBUS. On a 11/780 this would correspond to having the disk on "mba0" at "tr8" and the tape on "mba1" at "tr9". Here the MASSBUS adapter with the lowest TR number has been called "mba0" and the one with the next lowest number is called "mba1".

To find out the MASSBUS your tape and disk are on you can examine the cabling and the unit numbers or your site maintenance guide. Do not be fooled into thinking that the number on the front of the tape drive is a device number; it is a *slave* number, one of several possible tapes on the single tape formatter. For bootstrapping the slave number must be 0. The formatter unit number may be anything distinct from the other numbers on the same MASSBUS, but you must know what it is.

The MASSBUS devices are known by several different names by DEC software and by UNIX. At various times it is necessary to know both names. There is, of course, the name of the device like "RM03" or "RM80"; these are easy to remember because they are printed on the front of the device. DEC also gives devices names by the names of the driver in the system using a naming convention that reflects the interconnect topology of the machine. The first letter of such a name is a "D" for a disk, the second letter depends on the type of the drive, "DR" for RM03, RM05, and RM80's, "DB" for RP06's. The next letter is related to the interconnect; DEC calls the first MASSBUS adapter "A", the second "B", etc. Thus "DRA" is a RM drive on the first MASSBUS adapter. Finally, the name ends in a digit corresponding to the unit number for the device on the MASSBUS, i.e. "DRA0" is a disk at the first device slot on the first MASSBUS adapter and is a RM disk.

** References of the form X(Y) mean the subsection named X in section Y of the UNIX programmer's manual.

1.4. UNIX device naming

UNIX has a set of names for devices, which are different from the DEC names for the devices, viz.:

RM/RP disks	hp
TE/TU tapes	ht
TU78 tape	mt

The normal standalone system, used to bootstrap the full UNIX system, uses device names:

$xx(y,z)$

where *xx* is either **hp**, **ht**, or **mt**. The value *y* specifies the MASSBUS to use and also the device. It is computed as

$8 * mba + device$

Thus mba0 device 0 would have a *y* value of 0 while mba1 device 0 would have a *y* value of 8. The *z* value is interpreted differently for tapes and disks: for disks it is a disk *partition* (in the range 0-7), and for tapes it is a file number on the tape.

Each UNIX physical disk is divided into 8 logical disk partitions, each of which may occupy any consecutive cylinder range on the physical device. The cylinders occupied by the 8 partitions for each drive type are specified in section 4 of the programmers manual and in the disk description file /etc/disktab (c.f. *disktab*(5)).* Each partition may be used for either a raw data area such as a paging area or to store a UNIX file system. It is conventional for the first partition on a disk to be used to store a root file system, from which UNIX may be bootstrapped. The second partition is traditionally used as a paging area, and the rest of the disk is divided into spaces for additional "mounted file systems" by use of one or more additional partitions.

The third logical partition of each physical disk also has a conventional usage: it allows access to the entire physical device, including the bad sector forwarding information recorded at the end of the disk (one track plus 126 sectors). It is occasionally used to store a single large file system or to access the entire pack when making a copy of it on another. Care must be taken when using this partition to not overwrite the last few tracks and thereby clobber the bad sector information.

The disk partitions have names in the standalone system of the form "hp(x,y)" with varying *y* as described above. Thus partition 1 of a RM05 on mba0 at drive 0 would be "hp(0,1)". When not running standalone, this partition would normally be available as "/dev/hp0b". Here the prefix "/dev" is the name of the directory where all "special files" normally live, the "hp" serves an obvious purpose, the "0" identifies this as a partition of hp drive number "0" and the "b" identifies this as the first partition (where we number from 0, the 0'th partition being "hp0a".)

In all simple cases, a drive with unit number 0 (in its unit plug on the front of the drive) will be called unit 0 in its UNIX file name. This is not, however, strictly necessary, since the system has a level of indirection in this naming. This can be taken advantage of to make the system less dependent on the interconnect topology, and to make reconfiguration after hardware failure extremely easy. We will not discuss that now.

Returning to the discussion of the standalone system, we recall that tapes also took two integer parameters. In the normal case where the tape formatter is unit 0 on the second mba

---

* It is possible to change the partitions by changing the code for the table in the disk driver; since it is often desirable to do this it is clear that these tables should be read off each pack; they may be in a future version of the system.

(mba1), the files on the tape have names "ht(8,0)", "ht(8,1)", etc. Here "file" means a tape file containing a single data stream. The distribution tapes have data structures in the tape files and though the tapes contain only 6 tape files, they contain several thousand UNIX files.

For the UNIBUS, there are also conventional names. The important DEC names to know are DM?? for RK07 drives and DU?? for drives on a UDA50. For example, RK07 drive 0 on a controller on the first UNIBUS on the machine is "DMA0". UNIX calls such a device a "hk" and the standalone name for the first partition of such a device is "hk(0,0)". If the controller were on the second UNIBUS its name would be "hk(8,0)". If we wished to access the first partition of a RK07 drive 1 on uba0 we would use "hk(1,0)".

The UNIBUS disk and tape names used by UNIX are:

RK disks	hk
TS tapes	ts
UDA disks	ra
IDC disks	rb
SMD disks	up
TM tapes	tm
TU tapes	ut

Here SMD disks are disks on an RM emulating controller on the UNIBUS, and TM tapes are tapes on a controller that emulates the DEC TM-11. TU tapes are tapes on a controller that emulates the DEC TU45. IDC disks are disks on an 11/730 Integral Disk Controller. TS tapes are tapes on a controller that emulates the DEC TS-11 (e.g. a TU80). The naming conventions for partitions in UNIBUS disks and files in UNIBUS tapes are the same as those for MASSBUS disks and tapes.

1.5. UNIX devices: block and raw

UNIX makes a distinction between "block" and "raw" (character) devices. Each disk has a block device interface where the system makes the device byte addressable and you can write a single byte in the middle of the disk. The system will read out the data from the disk sector, insert the byte you gave it and put the modified data back. The disks with the names "/dev/xx0a", etc are block devices. There are also raw devices available. These have names like "/dev/rxx0a", the "r" here standing for "raw". In the bootstrap procedures we will often suggest using the raw devices, because these tend to work faster in some cases. In general, however, the block devices are used. They are where file systems are "mounted".

You should be aware that it is sometimes important to use the character device (for efficiency) or not (because it wouldn't work, e.g. to write a single byte in the middle of a sector). Don't change the instructions by using the wrong type of device indiscriminately.

September 22, 1983

2. BOOTSTRAP PROCEDURE

This section explains the bootstrap procedure that can be used to get the kernel supplied with this tape running on your machine. Even if you are currently running UNIX you will have to do a full bootstrap.

If you are already running UNIX you should first save your existing files on magnetic tape. 4.2BSD uses a totally different file system organization than previous versions of the system; it is thus necessary to rebuild the file system format before restoring the data. The easiest way to save the current files on tape is by doing a full dump and then restoring under the new system. Refer to chapter 3 in understanding how to upgrade an existing 4BSD system.

Booting from tape

The tape bootstrap procedure used to create a working system involves the following major steps:

1) Format a disk pack with the *format* program.

2) Copy a "mini root" file system from the tape onto the swap area of the disk.

3) Boot the UNIX system on the "mini root".

4) Restore the full root file system using *restore* (8).

5) Build a console floppy or cassette for bootstrapping.

6) Reboot the completed root file system.

7) Build and restore the /usr file system from tape with *tar* (1).

Certain of these steps are dependent on your hardware configuration. Formatting the disk pack used for the root file system may require using the DEC standard formatting programs. Also, if you are bootstrapping the system on an 11/750, no console cassette is created.

The following sections describe the above steps in detail. In these sections references to disk drives are of the form $xx(n,m)$ and references to files on tape drives are of the form $yy(n,m)$ where xx and yy are one of the names described in section 1.4 and n and m are the unit and offset numbers described in section 1.4. Commands you are expected to type are shown in roman, while that information printed by the system is shown emboldened. Throughout the installation steps the reboot switch on an 11/780 or 11/730 should be set to off; on an 11/750 set the power-on action to halt. (In normal operation an 11/780 or 11/730 will have the reboot switch on and an 11/750 will have the power-on action set to reboot/restart.)

If you encounter problems in following the instructions in this part of the document, refer to Appendix C for help in troubleshooting.

2.1. Step 1: formatting the disk

All disks used with 4.2BSD should be formatted to insure the proper handling of physically corrupted disk sectors. If you have DEC disk drives, you should use the standard DEC formatter to format your disks. If not, the *format* program included in the distribution, or a vendor supplied formatting program, may be used to format disks. The *format* program is capable of formatting any of the following supported distribution devices:

     EMULEX MASSBUS: AMPEX 300M, 330M, CDC 300M, FUJITSU 404M
     EMULEX SC-21V UNIBUS: AMPEX 300M, 330M, CDC 300M, FUJITSU 160M, 404M

If you have run a pre-4.1BSD version of UNIX on the packs you are planning to use for bootstrapping it is likely that the bad sector information on the packs has been destroyed, since it was accessible as normal data in the last several tracks of the disk. You should therefore run

the formatter again to make sure the information is valid.

On an 11/750, to use a disk pack as a bootstrap device, sectors 0 through 15, the disk sectors in the files "/vmunix" (the system image) and "/boot" (the program that loads the system image), and the file system indices that lead to these two files must not have any errors. On an 11/780 or 11/730, the "boot" program is loaded from the console medium and includes device drivers for the "hp" and "up" disks which perform ECC correction and bad sector forwarding; consequently, on these machines the system may be bootstrapped on these disks even if the disk is not error free in critical locations. In general, if the first 15884 sectors of your disk are clean you are safe; if not you can take your chances.

To load the *format* program, insert the distribution TU58 cassette or RX01 floppy disk in the appropriate console device (on the 11/730 use cassette 0) and perform the following steps.

If you have an 11/780 give the commands:

    >>> HALT
    >>> UNJAM
    >>> LOAD FORMAT
    >>> START 2

If you have an 11/750 give the commands:

    >>> I
    >>> B DDA0
    = format

If you have an 11/730 give the commands:

    >>> H
    >>> I
    >>> L DD0:FORMAT
    >>> S 2

The *format* program should now be running and awaiting your input:

Disk format/check utility

Enable debugging (1=bse, 2=ecc, 3=bse+ecc)?

If you made a mistake loading the program off the TU58 cassette the "=" prompt should reappear and you can retype the program name. If something else happened, you may have a bad distribution cassette or floppy, or your hardware may be broken; refer to Appendix C for help in troubleshooting. If you are unable to load programs off the distributed medium, consult Appendix B for an alternate (more painful) approach.

Format will create sector headers and verify the integrity of each sector formatted by using the disk controller's "write check" command. Remember *format* runs only on the **up** and **hp** drives indicated above. *Format* will prompt for the information required as shown below. If you make a mistake in answering questions, "#" erases the last character typed, and "@" erases the current input line.

> **Enable debugging (0=none, 1=bse, 2=ecc, 3=bse+ecc)?**
> **Device to format?** xx(0,0)
> ...(the old bad sector table is read; ignore any errors that occur here)...
> **Formatting drive** xx0 **on adaptor 0: verify (yes/no)?** *yes*
> **Device data: #cylinders=842, #tracks=20, #sectors=48**
> **Available test patterns are:**
>   **1 - (f00f) RH750 worst case**
>   **2 - (ec6d) media worst case**
>   **3 - (a5a5) alternating 1's and 0's**
>   **4 - (ffff) Severe burnin (takes several hours)**
> **Pattern (one of the above, other to restart)?** *2*
> **Start formatting...make sure the drive is online**
> *...(soft ecc's and other errors are reported as they occur)...*
> *...(if 4 write check errors were found, the program terminates like this)...*
> **Errors:**
> **Write check: 4**
> **Bad sector: 0**
> **ECC: 0**
> **Skip sector: 0**
> **Total of 4 hard errors found.**
> **Writing bad sector table at block 524256**
> *(524256 is the block # of the first block in the bad sector table)*
> **Done**

Once the root device has been formatted, *format* will prompt for another disk to format. Halt the machine by typing "control-P" and "H" (the "H" is necessary only on an 11/780, but does not hurt on the other machines).

> **Enable debugging (1=bse, 2=ecc, 3=bse+ecc)?**^P
> **>>>H**

It may be necessary to format other drives before constructing file systems on them; this can be done at a later time with the steps just performed. *Format* can also be used in an extended test mode (pattern 4) that uses numerous test patterns in 46 passes to detect as many disk surface errors as possible; this test runs for many hours, depending on the CPU and controller. On an 11/780, this can be speeded up significantly by setting the clock fast.

2.2. Step 2: copying the mini-root file system

The second step is to run a simple program, *copy*, which copies a very small root file system into the second partition of the disk. This file system will serve as the base for creating the actual root file system to be restored. The version of the operating system maintained on the "mini-root" file system understands not to swap on top of itself, thereby allowing double use of the disk partition. *Copy* is loaded just as the *format* program was loaded; for example, on an 11/780:

> (copy mini root file system)
> **>>>LOAD COPY**
> **>>>START 2**
> **From:** yy(y,1)   (unit *y*, second tape file)
> **To:** xx(x,1)   (mini root is on drive *x*, second partition)
> **Copy completed: 205 records copied**
> **From:**

while for an 11/750:

September 22, 1983

```
(copy mini root file system)
>>> B DDA0
= copy
From: yy(y,1)                     (unit y, second tape file)
To: xx(x,1)                       (mini root is on drive x, second partition)
Copy completed: 205 records copied
From:
```

and for an 11/730:

```
(copy mini root file system)
>>> L DD0:COPY
>>> S 2
From: yy(y,1)                     (unit y, second tape file)
To: xx(x,1)                       (mini root is on drive x, second partition)
Copy completed: 205 records copied
From:
```

(As above, '#' erases characters and '@' erases lines.)

2.3. Step 3: booting from the mini-root file system

You now have the minimal set of tools necessary to create a root file system and restore the file system contents from tape. To access this file system load the bootstrap program and boot the version of unix which has been placed in the "mini-root":

```
(load bootstrap program)
>>> LOAD BOOT
>>> START 2
Boot
: xx(x,1)vmunix                   (bring in vmunix off mini root)
```

or, on an 11/750:

```
(load bootstrap program)
>>> B DDA0
= boot
Boot
: xx(x,1)vmunix                   (bring in vmunix off mini root)
```

or, on an 11/730:

```
(load bootstrap program)
>>> L DD0:BOOT
>>> S 2
Boot
: xx(x,1)vmunix                   (bring in vmunix off mini root)
```

(As above, '#' erases characters and '@' erases lines.)

The standalone boot program should then read the system from the mini root file system you just created, and the system should boot:

September 22, 1983

```
215564+64088+69764 start 0xf98
4.2 BSD UNIX #1: Sun Feb 6 15:02:15 PST 1983
real mem = xxx
avail mem = yyy
... information about available devices ...
root device?
```

The first three numbers are printed out by the bootstrap programs and are the sizes of different parts of the system (text, initialized and uninitialized data). The system also allocates several system data structures after it starts running. The sizes of these structures are based on the amount of available memory and the maximum count of active users expected, as declared in a system configuration description. This will be discussed later.

UNIX itself then runs for the first time and begins by printing out a banner identifying the release and version of the system that is in use and the date it was compiled.

Next the *mem* messages give the amount of real (physical) memory and the memory available to user programs in bytes. For example, if your machine has only 512K bytes of memory, then xxx will be 523264, 1024 bytes less than 512K. The system reserves the last 1024 bytes of memory for use in error logging and doesn't count it as part of real memory.

The messages that came out next show what devices were found on the current processor. These messages are described in *autoconf*(4). The distributed system may not have found all the communications devices you have (dh's and dz's), or all the mass storage peripherals you have if you have more than two of anything. This will be corrected soon, when you create a description of your machine to configure UNIX from. The messages printed at boot here contain much of the information that will be used in creating the configuration. In a correctly configured system most of the information present in the configuration description is printed out at boot time as the system verifies that each device is present.

The "root device?" prompt was printed by the system and is now asking you for the name of the root file system to use. This happens because the distribution system is a *generic* system. It can be bootstrapped on any VAX cpu and with its root device and paging area on any available disk drive. You should respond to the root device question with xx0*. This response supplies two pieces of information: first, xx0 indicates the disk it is running on is drive 0 of type xx, secondly the "*" indicates the system is running "atop" the paging area. The latter is most important, otherwise the system will attempt to page on top of itself and chaos will ensue. You will later build a system tailored to your configuration that will not ask this question when it is bootstrapped.

```
root device? xx0*
WARNING: preposterous time in file system − − CHECK AND RESET THE DATE!
erase ^?, kill ^U, intr ^C
#
```

The "erase ..." message is part of /.profile that was executed by the root shell when it started. This message is present to remind you that the line character erase, line erase, and interrupt characters are set to be what is standard on DEC systems; this insures things are consistent with the DEC console interface characters.

2.4. Step 4: restoring the root file system

UNIX is now running, and the 'UNIX Programmer's manual' applies. The '#' is the prompt from the shell, and lets you know that you are the super-user, whose login name is "root". To complete installation of the bootstrap system two steps remain. First, the root file system must be created, and second a boot floppy or cassette must be constructed.

To create the root file system the shell script "xtr" should be run as follows:

```
#disk=xx0  type=tt  tape=yy  xtr
```

where *xx0* is the name of the disk on which the root file system is to be restored (unit 0), *tt* is the *tt* of drive on which the root file system is to be restored (see the table below), and *yy* is the name of the tape drive on which the distribution tape is mounted.

If the root file system is to reside on a disk other than unit 0 (as shown in the information printed out during autoconfiguration), you will have to create the necessary special files in /dev and use the appropriate value. For example, if the root should be placed on hp1, you must create /dev/rhp1a and /dev/hp1a using *mknod*(8).

Drive	Type	Drive	Type
DEC RM03	type=rm03	DEC RM05	type=rm05
DEC RM80	type=rm80	DEC RP06	type=rp06
DEC RP07	type=rp07	DEC RK07	type=rk07
DEC RA80	type=ra80	DEC RA60	type=ra60
DEC RA81	type=ra81	DEC R80	type=rb80
CDC 9766	type=9766	CDC 9775	type=9775
AMPEX 300M	type=9300	AMPEX 330M	type=capricorn
FUJITSU 160M	type=fuji160	FUJITSU 404M	type=eagle

This will generate many messages regarding the construction of the file system and the restoration of the tape contents, but should eventually terminate with the messages:

```
...
Root filesystem extracted

If this is a 780, update floppy
If this is a 730, update the cassette
#
```

2.5. Step 5: creating a boot floppy or cassette

If the machine is an 11/780 or 11/730, a boot floppy or cassette should be constructed according to the instructions in chapter 4. For 11/750's, bootstrapping is performed by using a boot prom and special code located in sectors 0-15 of the root file system. The *newfs* program automatically installs the needed code, so you may continue on to the next step. On an 11/780 with interleaved memory, or other configurations that require alteration of the standard boot files, this step may be left for later.

2.6. Step 6: rebooting the completed root file system

With the above work completed, all that is left is to reboot:

September 22, 1983

```
#sync                                    (synchronize file system state)
#^P                                      (halt machine)
>>>HALT                                  (for 11/780's only)
>>>UNJAM                                 (for 11/780's only)
>>>I                                     (initialize processor state)
>>>B xxS                                 (on an 11/750, use B/2)
...(boot program is eventually loaded)...
Boot
: xx(x,0)vmunix                          (vmunix brought in off root)
215564+64088+69764 start 0xf98
4.2 BSD UNIX #1: Sun Feb 6 15:02:15 PST 1983
real mem = xxx
avail mem = yyy
... information about available devices ...
root on xx0
WARNING: preposterous time in file system — CHECK AND RESET THE DATE!
erase ^?, kill ^U, intr ^C
#
```

(see section 6.1 if the system does not reboot properly)

The system is now running single user on the installed root file system. The next section tells how to complete the installation of distributed software on the /usr file system.

2.7. Step 7: setting up the /usr file system

First set a shell variable to the name of your disk, so the commands we give will work regardless of the disk you have; do one of

```
# disk=hp    (if you have an RP06, RM03, RM05, RM80, or other MASSBUS drive)
# disk=hk    (if you have RK07s)
# disk=ra    (if you have UDA50 storage module drives)
# disk=up    (if you have UNIBUS storage module drives)
# disk=rb    (if you have IDC storage module drives)
```

The next thing to do is to extract the rest of the data from the tape. You might wish to review the disk configuration information in section 4.4 before continuing; the partitions used below are those most appropriate in size. Find the disk you have in the following table and execute the commands in the right hand portion of the table:

September 22, 1983

```
DEC RM03                # name=hp0g; type=rm03
DEC RM05                # name=hp0g; type=rm05
DEC RM80                # name=hp0g; type=rm80
DEC RP06                # name=hp0g; type=rp06
DEC RP07                # name=hp0h; type=rp07
DEC RK07                # name=hk0g; type=rk07
DEC RA80                # name=ra0h; type=ra80
DEC RA60                # name=ra0h; type=ra60
DEC RA81                # name=ra0h; type=ra81
DEC R80                 # name=rb0h; type=rb80
UNIBUS CDC 9766         # name=up0g; type=9766
UNIBUS AMPEX 300M       # name=up0g; type=9300
UNIBUS AMPEX 330M       # name=up0g; type=capricorn
UNIBUS FUJITSU 160M     # name=up0g; type=fuji160
UNIBUS FUJITSU 404M     # name=up0h; type=eagle
MASSBUS CDC 9766        # name=hp0g; type=9766
MASSBUS AMPEX 300M      # name=hp0g; type=9300
MASSBUS AMPEX 330M      # name=hp0g; type=capricorn
MASSBUS FUJITSU 404M    # name=hp0h; type=eagle
```

Find the tape you have in the following table and execute the commands in the right hand portion of the table:

```
DEC TE16/TU45/TU77      # cd /dev; MAKEDEV ht0; sync
DEC TU78                # cd /dev; MAKEDEV mt0; sync
DEC TS11                # cd /dev; MAKEDEV ts0; sync
EMULEX TC11             # cd /dev; MAKEDEV tm0; sync
SI 9700                 # cd /dev; MAKEDEV ut0; sync
```

Then execute the following commands

```
# date yymmddhhmm              (set date, see date (1))
....
# passwd root                  (set password for super-user)
New password:                  (password will not echo)
Retype new password:
# newfs ${name} ${type}        (create empty user file system)
(this takes a few minutes)
# mount /dev/${name} /usr      (mount the usr file system)
# cd /usr                      (make /usr the current directory)
# mkdir sys                    (make directory for system source)
# cd sys                       (make /usr/sys the current directory)
# mt fsf
# tar xpbf 20 /dev/rmt12       (extract the system source)
(this takes about 5-10 minutes)
# cd ..                        (back to /usr)
# mt fsf
# tar xpbf 20 /dev/rmt12       (extract all of usr except usr/src)
(this takes about 15-20 minutes)
# cd /                         (back to root)
# chmod 755 /  /usr  /usr/sys
# rm −f sys
# ln −s /usr/sys sys           (make a symbolic link to the system source)
# umount /dev/${name}          (unmount /usr)
```

The data on the fourth and fifth tape files has now been extracted and the first reel of the distribution is no longer needed. The remainder of the installation procedure uses the second reel of tape which should be mounted in place of the first.

You can check the consistency of the /usr file system by doing

```
# fsck /dev/r${name}
```

The output from *fsck* should look something like:

** /dev/rxx0h
** Last Mounted on /usr
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Cyl groups
671 files, 3497 used, 137067 free (75 frags, 34248 blocks)

If there are inconsistencies in the file system, you may be prompted to apply corrective action; see the document describing *fsck* for information.

To use the /usr file system, you should now remount it by saying

```
# /etc/mount /dev/${name} /usr
```

You can now extract the first file on the second tape (the source for the commands). If you have RK07's you must first put a formatted pack in drive 1 and set up a UNIX file system on it by doing:

September 22, 1983

```
# newfs hk1g rk07
(this takes a few minutes)
# mount /dev/hk1g /usr/src
# cd /usr/src
```

In any case you can then extract the source code for the commands (except on RK07's this will fit in the /usr file system):

```
# mkdir /usr/src
# chmod 755 /usr/src
# cd /usr/src
# tar xpb 20
```

If you get an error at this point, you can reposition the tape with the following command and try the above commands again.

```
# mt rew
```

2.8. Additional software

There are three extra tape files on the distribution tapes which have not been installed to this point. They are a font library for use with Varian and Versatec printers, the Ingres database system, and user contributed software. All three tapes files are in *tar*(1) format and can be installed by positioning the tape and reading in the files as was done for /usr/src above. As distributed, the fonts should be placed in a directory /usr/lib/vfont, the Ingres system should be placed in /usr/ingres, and the user contributed software should be placed in /usr/src/new. The exact contents of the user contributed software is given in a separate document.

September 22, 1983

3. UPGRADING A 4BSD SYSTEM

Begin by reading the other parts of this document to see what has changed since the last time you bootstrapped the system. Also read the "Changes in 4.2BSD" document, and look at the new manual sections provided to you. If you have local system modifications to the kernel to install, look at the document "Kernel changes in 4.2BSD" to get an idea of how the system changes will affect your local mods.

If you are running a version of the system distributed prior to 4.0BSD, you are pretty much on your own. Sites running 3BSD or 32/V may be able to modify the restor program to understand the old 512 byte block file system, but this has never been tried. This section assumes you are running 4.1BSD.

3.1. Step 1: what to save

No matter what version of the system you may be running, you will have to rebuild your root and usr file systems. The easist way to do this is to save the important files on your existing system, perform a bootstrap as if you were installing 4.2BSD on a brand new machine, then merge the saved files into the new system. The following list enumerates the standard set of files you will want to save and indicates directories in which site specific files should be present. This list will likely be augmented with non-standard files you have added to your system; be sure to do a tar of the directories /etc, /lib, and /usr/lib to guard against your missing something the first time around.

/.profile	root sh startup script
/.login	root csh startup script
/.cshrc	root csh startup script
/dev/MAKE	for the LOCAL case for making devices
/etc/fstab	disk configuration data
/etc/group	group data base
/etc/passwd	user data base
/etc/rc	for any local additions
/etc/ttys	terminal line configuration data
/etc/ttytype	terminal line to terminal type mapping data
/etc/termcap	for any local entries which may have been added
/lib	for any locally developed language processors
/usr/dict/*	for local additions to words and papers
/usr/include/*	for local additions
/usr/lib/aliases	mail forwarding data base
/usr/lib/crontab	cron daemon data base
/usr/lib/font/*	for locally developed font libraries
/usr/lib/lint/*	for locally developed lint libraries
/usr/lib/tabset/*	for locally developed tab setting files
/usr/lib/term/*	for locally developed nroff drive tables
/usr/lib/tmac/*	for locally developed troff/nroff macros
/usr/lib/uucp/*	for local uucp configuration files
/usr/man/manl	for manual pages for locally developed programs
/usr/msgs	for current msgs
/usr/spool/*	for current mail, news, uucp files, etc.
/usr/src/local	for source for locally developed programs

As 4.1BSD binary images will run unchanged under 4.2BSD you should be certain to save any programs such as compilers which you will need in bootstrapping to 4.2BSD.*

---

* 4.2BSD can support a "4.1BSD compatibility mode" of system operation whereby system calls from 4.1BSD

September 22, 1983

Once you have saved the appropriate files in a convenient format, the next step is to dump your file systems with *dump*(8). For the utmost of safety this should be done to magtape. However, if you enjoy gambling with your life (or you have a VERY friendly user community) and you have sufficient disk space, you can try converting your file systems in-place by using a disk partition. If you select the latter tact, a version of the 4.1BSD dump program which runs under 4.2 is provided in /etc/dump.4.1; be sure to read through this entire document before beginning the conversion. Beware that file systems created under 4.2BSD will use about 5-10% more disk space for file system related information than under 4.1BSD. Thus, before dumping each file system it is a good idea to remove any files which may be easily regenerated. Since most all programs will likely be recompiled under the new system your best bet is to remove any object files. File systems with at least 10% free space on them should restore into an equivalently sized 4.2BSD file system without problem.

Once you have dumped the file systems you wish to convert to 4.2BSD, install the system from the bootstrap tape as described in chapter 2, then proceed to the next section.

3.2. Step 2: merging

When your system is booting reliably and you have the 4.2BSD root and /usr file systems fully installed you will be ready to proceed to the next step in the conversion process: merging your old files into the new system.

Using the tar tape, or tapes, you created in step 1 extract the appropriate files into a scratch directory, say /usr/convert:

```
# mkdir /usr/convert
# cd /usr/convert
# tar x
```

Certain data files, such as those from the /etc directory, may simply be copied into place.

```
# cp passwd group fstab ttys ttytype /etc
# cp crontab /usr/lib
```

Other files, however, must be merged into the distributed versions by hand. In particular, be careful with /etc/termcap.

The commands kept under the LOCAL entry in /dev/MAKE should be placed in the new shell script /dev/MAKEDEV.local so that saying "MAKEDEV LOCAL" will create the appropriate local devices and device names. If you have any homegrown device drivers which use major device numbers reserved by the system you will have to modify the commands used to create the devices or alter the system device configuration tables in /sys/vax/conf.c.

The spooling directories saved on tape may be restored in their eventual resting places without too much concern. Be sure to use the 'p' option to tar so that files are recreated with the same file modes:

```
# cd /usr
# tar xp msgs spool/mail spool/uucp spool/uucppublic spool/news
```

Whatever else is left is likely to be site specific or require careful scrutiny before placing in its eventual resting place. Refer to the documentation and source code before arbitrarily overwriting a file.

---

are either emulated or safely ignored. There are only two exceptions; programs which read directories or use the old jobs library will not operate properly. However, while 4.1BSD binaries will execute under 4.2BSD it is **STRONGLY RECOMMENDED** that the programs be recompiled under the new system. Refer to the document "Changes in 4.2BSD" for elaboration on this point.

3.3. Step 3: converting file systems

The dump format used in 4.0 and 4.1BSD is upward compatible with that used in 4.2BSD. That is, the 4.2BSD *restore* program understands how to read old dump tapes, although 4.2BSD dump tapes may not be properly restored under 4.0BSD or 4.1BSD. To convert a file system dumped to magtape, simply create the appropriate file system and restore the data. Note that the 4.2BSD *restore* program does its work on a mounted file system using normal system operations (unlike the older *restor* which accessed the raw file system device and deposited inodes in the appropriate locations on disk). This means that file system dumps may be restored even if the characteristics of the file system changed. To restore a dump tape for, say, the /a file system something like the following would be used:

```
# mkdir /a
# newfs hp1g eagle
# mount /dev/hp1g /a
# cd /a
# restore r
```

If tar images were written instead of doing a dump, you should be sure to use the 'p' option when reading the files back. No matter how you restore a file system, be sure and check its integrity with fsck when the job is complete.

3.4. Bootstrapping language processors

To convert a compiler from 4.1BSD to 4.2BSD you should simply have to recompile and relink the various parts. If the processor is written in itself, for instance a PASCAL compiler written in PASCAL, the important step in converting is to save a working copy of the 4.1BSD binary before converting to 4.2BSD. Then, once the system has been changed over, the 4.1BSD binary should be used in the rebuilding process. In order to do this, you should enable the 4.1 compatibility option when you configure the kernel (below).

If no working 4.1BSD binary exists, or the language processor uses some nonstandard system call, you will likely have to compile the language processor into an intermediate form, such as assembly language, on a 4.1BSD system, then bring the intermediate form to 4.2BSD for assembly and loading.

4. SYSTEM SETUP

This section describes procedures used to setup a VAX UNIX system. Procedures described here are used when a system is first installed or when the system configuration changes. Procedures for normal system operation are described in the next section.

4.1. Making a UNIX boot floppy

If you have an 11/780 you will want to create a UNIX boot floppy by adding some files to a copy of your current DEC console floppy, using *flcopy*(8) and *arff*(8). This floppy will make standalone system operations such as bootstrapping much easier.

First change into the directory where the console floppy information is stored:

    # cd /sys/floppy

then set up the default boot device. If you have an RK07 as your primary root do:

    # cp defboo.hk defboo.cmd

If you have a drive on a UDA50 (e.g. an RA81) as your primary root do:

    # cp defboo.ra defboo.cmd

If you have a second vendor UNIBUS storage module as your primary root do:

    # cp defboo.up defboo.cmd

Otherwise:

    # cp defboo.hp defboo.cmd

If the local configuration requires any changes in restar.cmd or defboo.cmd (e.g., for inter-leaved memory controllers), these should be made now. The following command will then copy your DEC local console floppy, updating the copy appropriately.

    # make update
    Change Floppy, Hit return when done.
    (waits for you to put clean floppy in console)
    Are you sure you want to clobber the floppy? yes

More copies of this floppy can be made using *flcopy*(8).

4.2. Making a UNIX boot cassette

If you have an 11/730 you will want to create a UNIX boot cassette by adding some files to a copy of your current DEC console cassette, using *flcopy*(8) and *arff*(8). This cassette will make standalone system operations such as bootstrapping much easier.

First change into the directory where the console cassette information is stored:

    # cd /sys/cassette

then set up the default boot device. If you have an IDC storage module as your primary root do:

    # cp defboo.rb defboo.cmd

If you have an RK07 as your primary root do:

    # cp defboo.hk defboo.cmd

If you have a drive on a UDA50 as your primary root do:

    # cp defboo.ra defboo.cmd

September 22, 1983

Otherwise:

>   # cp defboo.up defboo.cmd

To complete the procedure place your DEC local console cassette in drive 0 (the drive at front of the CPU); the following command will then copy it, updating the copy appropriately.

>   # make update
>   **Change Floppy, Hit return when done.**
>   (waits for you to put clean cassette in console drive 0)
>   **Are you sure you want to clobber the floppy?** yes

More copies of this cassette can best be made using *dd* (1).

4.3. Kernel configuration

This section briefly describes the layout of the kernel code and how files for devices are made. For a full discussion of configuring and building system images, consult the document "Building 4.2BSD UNIX Systems with Config".

4.3.1. Kernel organization

As distributed, the kernel source is in a separate tar image. The source may be physically located anywhere within any file system so long as a symbolic link to the location is created for the file /sys (many files in /usr/include are normally symbolic links relative to /sys). In further discussions of the system source all path names will be given relative to /sys.

The directory /sys/sys contains the mainline machine independent operating system code. Files within this directory are conventionally named with the following prefixes.

init_	system initialization
kern_	kernel (authentication, process management, etc.)
quota_	disk quotas
sys_	system calls and similar
tty_	terminal handling
ufs_	file system
uipc_	interprocess communication
vm_	virtual memory

The remaining directories are organized as follows.

/sys/h	machine independent include files
/sys/conf	site configuration files and basic templates
/sys/net	network independent, but network related code
/sys/netinet	DARPA Internet code
/sys/netimp	IMP support code
/sys/netpup	PUP-1 support code
/sys/vax	VAX specific mainline code
/sys/vaxif	VAX network interface code
/sys/vaxmba	VAX MASSBUS device drivers and related code
/sys/vaxuba	VAX UNIBUS device drivers and related code

Many of these directories are referenced through /usr/include with symbolic links. For example, /usr/include/sys is a symbolic link to /sys/h. The system code, as distributed, is totally independent of the include files in /usr/include. This allows the system to be recompiled from scratch without the /usr file system mounted.

September 22, 1983

4.3.2. Devices and device drivers

Devices supported by UNIX are implemented in the kernel by drivers whose source is kept in /sys/vax, /sys/vaxuba, or /sys/vaxmba. These drivers are loaded into the system when included in a cpu specific configuration file kept in the conf directory. Devices are accessed through special files in the file system, made by the *mknod*(8) program and normally kept in the /dev directory. For all the devices supported by the distribution system, the files in /dev are created by the /dev/MAKEDEV shell script.

Determine the set of devices that you have and create a new /dev directory by running the MAKEDEV script. First create a new directory /newdev, copy MAKEDEV into it, edit the file MAKEDEV.local to provide an entry for local needs, and run it to generate a /newdev directory. For instance, if your machine has a single dz-11, a single dh-11, a single dmf-32, an rm03 disk, an EMULEX controller, an AMPEX-9300 disk, and a te16 tape drive you would do:

```
# cd /
# mkdir newdev
# cp dev/MAKEDEV newdev/MAKEDEV
# cd newdev
# MAKEDEV dz0 dh0 dmf0 hp0 up0 ht0 std LOCAL
```

Note the "std" argument causes standard devices such as /dev/console, the machine console, /dev/floppy, the console floppy disk interface for the 11/780, and /dev/tu0 and /dev/tu1, the console cassette interfaces for the 11/750 and 11/730, to be created.

You can then do

```
# cd /
# mv dev olddev ; mv newdev dev
# sync
```

to install the new device directory.

4.3.3. Building new system images

The kernel configuration of each UNIX system is described by a single configuration file, stored in the /sys/conf directory. To learn about the format of this file and the procedure used to build system images, start by reading "Building 4.2BSD UNIX Systems with Config", look at the manual pages in section 4 of the UNIX manual for the devices you have, and look at the configuration files in the /sys/conf directory.

The configured system image "vmunix" should be copied to the root, and then booted to try it out. It is best to name it /newvmunix so as not to destroy the working system until you're sure it does work:

```
# cp vmunix /newvmunix
# sync
```

It is also a good idea to keep the old system around under some other name. In particular, we recommend that you save the generic distribution version of the system permanently as /genvmunix for use in emergencies.

To boot the new version of the system you should follow the bootstrap procedures outlined in section 6.1. A systematic scheme for numbering and saving old versions of the system is best.

4.4. Disk configuration

This section describes how to layout file systems to make use of the available space and to balance disk load for better system performance.

September 22, 1983

4.4.1. Initializing /etc/fstab

Change into the directory /etc and copy the appropriate file from:

      fstab.rm03
      fstab.rm05
      fstab.rm80
      fstab.ra60
      fstab.ra80
      fstab.ra81
      fstab.rb80
      fstab.rp06
      fstab.rp07
      fstab.rk07
      fstab.up160m (160Mb up drives)
      fstab.up300m (300Mb up drives)
      fstab.hp400m (400Mb hp drives)
      fstab.up (other up drives)
      fstab.hp (other hp drives)

to the file /etc/fstab, i.e.:

      # cd /etc
      # cp *fstab.xxx* fstab

This will set up the initial information about the usage of disk partitions, which we see how to update more below.

4.4.2. Disk naming and divisions

Each physical disk drive can be divided into up to 8 partitions; UNIX typically uses only 3 or 4 partitions. For instance, on an RM03 or RP06, the first partition, hp0a, is used for a root file system, a backup thereof, or a small file system like, /tmp; the second partition, hp0b, is used for paging and swapping; and the third partition hp0g holds a user file system. On an RM05, the first three partitions are used as for the RM03, and the fourth partition, hp0h, is used to hold the /usr file system, including source code.

The disk partition sizes for a drive are based on a set of four default partition tables; c.f. *diskpart* (8). The particular table used is dependent on the size of the drive. The "a" partition is the same size across all drives, 15884 sectors. The "b" partition, used for paging and swapping, is sized according to the total space on the disk. For drives less than about 400 megabytes the partition is 33440 sectors, while for larger drives the partition size is doubled to 66880 sectors. The "c" partition is always used to access the entire physical disk, including the space at the back of the disk reserved for the bad sector forwarding table. If the disk is larger than about 250 megabytes, an "h" partition is created with size 291346 sectors, and no matter whether the "h" partition is created or not, the remainder of the drive is allocated to the "g" partition. Sites which want to split up the "g" partition into a number of smaller file systems may use the "d", "e", and "f" partitions which overlap the "g" partition. The default sizes for these partitions are 15884, 55936, and the remainder of the disk, respectively[*].

4.4.3. Space available

The space available on a disk varies per device. The amount of space available on the common disk partitions is listed in the following table. Not shown in the table are the partitions of each drive devoted to the root file system and the paging area.

---

[*] These rules are, unfortunately not evenly applied to all disks. Drives on DEC UDA50 and IDC controllers do not completely follow these rules; in particular, the swap partition on an RA81 is only 33440 sectors, and no "d", "e", or "f" partitions are available on an RA60 or RA80. Consult *uda* (4) for more information.

Type	Name	Size	Name	Size
rk07	hk?g	13 Mb		
rm03	hp?g	41 Mb		
rp06	hp?g	145 Mb		
rm05	hp?g	80 Mb	hp?h	145 Mb
rm80	hp?g	96 Mb		
ra60	ra?g	41 Mb	ra?h	139 Mb
ra80	ra?g	41 Mb	ra?h	56 Mb
ra81	ra?g	41 Mb	ra?h	380 Mb
rb80	rb?g	41 Mb	rb?h	56 Mb
rp07	hp?g	315 Mb	hp?h	145 Mb
up300	up?g	80 Mb	up?h	145 Mb
hp400	hp?g	216 Mb	hp?h	145 Mb
up160	up?g	106 Mb		

Here up300 refers to either an AMPEX or CDC 300 Megabyte disk on a UNIBUS disk controller, up160 refers to a FUJITSU 160 Megabyte disk on the UNIBUS, and hp400 refers to a FUJITSU Eagle 400 Megabyte disk on a MASBUS disk controller. Consult the manual pages for the specific controllers for other supported disks or other partitions.

Each disk also has a paging area, typically of 16 Megabytes, and a root file sytem of 8 Megabytes. The distributed system binaries occupy about 22 Megabytes while the major sources occupy another 25 Megabytes. This overflows dual RK07 and dual RL02 systems, but fits easily on most other hardware configurations.

Be aware that the disks have their sizes measured in disk sectors (512 bytes), while the UNIX file system blocks are variable sized. All user programs report disk space in kilobytes and, where needed, disk sizes are always specified in terms of sectors. The /etc/disktab file used in making file systems specifies disk partition sizes in sectors; the default sector size of 512 bytes may be overridden with the "se" attribute.

4.4.4. Layout considerations

There are several considerations in deciding how to adjust the arrangement of things on your disks: the most important is making sure there is adequate space for what is required; secondarily, throughput should be maximized. Paging space is an important parameter. The system, as distributed, sizes the configured paging areas each time the system is booted. Further, multiple paging areas of different size may be interleaved. Drives smaller than 400 megabytes have swap partitions of 16 megabytes while drives larger than 400 megabytes have 32 megabytes. These values may be changed to get more paging space by changing the appropriate partition table in the disk driver.

Many common system programs (C, the editor, the assembler etc.) create intermediate files in the /tmp directory, so the file system where this is stored also should be made large enough to accommodate most high-water marks; if you have several disks, it makes sense to mount this in a "root" (i.e. first partition) file system on another disk. All the programs that create files in /tmp take care to delete them, but are not immune to rare events and can leave dregs. The directory should be examined every so often and the old files deleted.

The efficiency with which UNIX is able to use the CPU is often strongly affected by the configuration of disk controllers. For general time-sharing applications, the best strategy is to try to split the root file system (/), system binaries (/usr), the temporary files (/tmp), and the user files among several disk arms, and to interleave the paging activity among a several arms.

It is critical for good performance to balance disk load. There are at least five components of the disk load that you can divide between the available disks:

1. The root file system.
2. The /tmp file system.
3. The /usr file system.
4. The user files.
5. The paging activity.

The following possibilities are ones we have used at times when we had 2, 3 and 4 disks:

	disks		
what	2	3	4
/	1	2	2
tmp	1	3	4
usr	1	1	1
paging	1+2	1+3	1+3+4
users	2	2+3	2+3
archive	x	x	4

The most important things to consider are to even out the disk load as much as possible, and to do this by decoupling file systems (on separate arms) between which heavy copying occurs. Note that a long term average balanced load is not important... it is much more important to have instantaneously balanced load when the system is busy.

Intelligent experimentation with a few file system arrangements can pay off in much improved performance. It is particularly easy to move the root, the /tmp file system and the paging areas. Place the user files and the /usr directory as space needs dictate and experiment with the other, more easily moved file systems.

4.4.5. File system parameters

Each file system is parameterized according to its block size, fragment size, and the disk geometry characteristics of the medium on which it resides. Inaccurate specification of the disk characteristics or haphazard choice of the file system parameters can result in substantial throughput degradation or significant waste of disk space. As distributed, file systems are configured according to the following table.

File system	Block size	Fragment size
/	8 Kbytes	1 Kbytes
usr	4 Kbytes	512 bytes
users	4 Kbytes	1 Kbytes

The root file system block size is made large to optimize bandwidth to the associated disk; this is particularly important since the /tmp directory is normally part of the root file. The large block size is also important as many of the most heavily used programs are demand paged out of the /bin directory. The fragment size of 1 Kbytes is a "nominal" value to use with a file system. With a 1 Kbyte fragment size disk space utilization is approximately the same as with the earlier versions of the file system.

The usr file system uses a 4 Kbyte block size with 512 byte fragment size in an effort to get high performance while conserving the amount of space wasted by a large fragment size. Space compaction has been deemed important here because the source code for the system is normally placed on this file system.

The file systems for users have a 4 Kbyte block size with 1 Kbyte fragment size. These parameters have been selected based on observations of the performance of our user file systems. The 4 Kbyte block size provides adequate bandwidth while the 1 Kbyte fragment size provides acceptable space compaction and disk fragmentation.

September 22, 1983

Other parameters may be chosen in constructing file systems, but the factors involved in choosing a block size and fragment size are many and interact in complex ways. Larger block sizes result in better throughput to large files in the file system as larger i/o requests will then be performed by the system. However, consideration must be given to the average file sizes found in the file system and the performance of the internal system buffer cache. The system currently provides space in the inode for 12 direct block pointers, 1 single indirect block pointer, and 1 double indirect block pointer.* If a file uses only direct blocks, access time to it will be optimized by maximizing the block size. If a file spills over into an indirect block, increasing the block size of the file system may decrease the amount of space used by eliminating the need to allocate an indirect block. However, if the block size is increased and an indirect block is still required, then more disk space will be used by the file because indirect blocks are allocated according to the block size of the file system.

In selecting a fragment size for a file system, at least two considerations should be given. The major performance tradeoffs observed are between an 8 Kbyte block file system and a 4 Kbyte block file system. Due to implementation constraints, the block size / fragment size ratio can not be greater than 8. This means that an 8 Kbyte file system will always have a fragment size of at least 1 Kbytes. If a file system is created with a 4 Kbyte block size and a 1 Kbyte fragment size, then upgraded to an 8 Kbyte block size and 1 Kbyte fragment size, identical space compaction will be observed. However, if a file system has a 4 Kbyte block size and 512 byte fragment size, converting it to an 8K/1K file system will result in significantly more space being used. This implies that 4 Kbyte block file systems which might be upgraded to 8 Kbyte blocks for higher performance should use fragment sizes of at least 1 Kbytes to minimize the amount of work required in conversion.

A second, more important, consideration when selecting the fragment size for a file system is the level of fragmentation on the disk. With a 512 byte fragment size, storage fragmentation occurs much sooner, particularly with a busy file system running near full capacity. By comparison, the level of fragmentation in a 1 Kbyte fragment file system is an order of magnitude less severe. This means that on file systems where many files are created and deleted the 512 byte fragment size is more likely to result in apparent space exhaustion due to fragmentation. That is, when the file system is nearly full, file expansion which requires locating a contiguous area of disk space is more likely to fail on a 512 byte file system than on a 1 Kbyte file system. To minimize fragmentation problems of this sort, a parameter in the super block specifies a minimum acceptable free space threshhold. When normal users (i.e. anyone but the super-user) attempt to allocate disk space and the free space threshold is exceeded, the user is returned an error as if the file system were actually full. This parameter is nominally set to 10%; it may be changed by supplying a parameter to *newfs*, or by patching the super block of an existing file system.

In general, unless a file system is to be used for a special purpose application (for example, storing image processing data), we recommend using the default values supplied. Remember that the current implementation limits the block size to at most 8 Kbytes and the ratio of block size / fragment size must be in the range 1-8.

The disk geometry information used by the file system affects the block layout policies employed. The file /etc/disktab, as supplied, contains the data for most all drives supported by the system. When constructing a file system you should use the *newfs* (8) program and specify the type of disk on which the file system resides. This file also contains the default file system partition sizes, and default block and fragment sizes. To override any of the default values you can modify the file or use one of the options to *newfs*.

---

* A triple indirect block pointer is also reserved, but not currently supported.

September 22, 1983

4.4.6. Implementing a layout

To put a chosen disk layout into effect, you should use the *newfs* (8) command to create each new file system. Each file system must also be added to the file /etc/fstab so that it will be checked and mounted when the system is bootstrapped.

As an example, consider a system with rm03's. On the first rm03, hp0, we will put the root file system in hp0a, and the /usr file system in hp0g, which has enough space to hold it and then some. The /tmp directory will be part of the root file system, as no file system will be mounted on /tmp. If we had only one rm03, we would put user files in the hp0g partition with the system source and binaries.

If we had a second rm03, we would create a file system in hp1g and put user files there, calling the file system /mnt. We would also interleave the paging between the 2 rm03's. To do this we would build a system configuration that specified:

        config   vmunix   root on hp0 swap on hp0 and hp1

to get the swap interleaved, and add the lines

        /dev/hp1b::sw::
        /dev/hp1g:/mnt:rw:1:2

to the /etc/fstab file. We would keep a backup copy of the root file system in the **hp1a** disk partition.

To make the /mnt file system we would do:

        # cd /dev
        # MAKEDEV hp1
        # newfs hp1g rm03
        (information about file system prints out)
        # mkdir /mnt
        # mount /dev/hp1g /mnt

4.5. Configuring terminals

If UNIX is to support simultaneous access from more than just the console terminal, the file /etc/ttys (*ttys* (5)) has to be edited.

Terminals connected via dz interfaces are conventionally named ttyDD where DD is a decimal number, the "minor device" number. The lines on dz0 are named /dev/tty00, /dev/tty01, ... /dev/tty07. Lines on dh or dmf interfaces are conventionally named ttyhX, where X is a hexadecimal digit. If more than one dh or dmf interface is present in a configuration, successive terminals would be named ttyiX, ttyjX, etc.

To add a new terminal, be sure the device is configured into the system and that the special file for the device has been made by /dev/MAKEDEV. Then, set the first character of the appropriate line of /etc/ttys to 1 (or add a new line).

The second character of each line in the /etc/ttys file lists the speed and initial parameter settings for the terminal. The commonly used choices are:

        0   300-1200-150-110
        2   9600
        3   1200-300
        5   300-1200

Here the first speed is the speed a terminal starts at, and "break" switches speeds. Thus a newly added terminal /dev/tty00 could be added as

        12tty00

if it was wired to run at 9600 baud. The definition of each "terminal type" is located in the file

September 22, 1983

/etc/gettytab and read by the *getty* program. To make custom terminal types, consult *gettytab*(5) before modifying this file.

Dialup terminals should be wired so that carrier is asserted only when the phone line is dialed up. For non-dialup terminals from which modem control is not available, you must either wire back the signals so that the carrier appears to always be present, or show in the system configuration that carrier is to be assumed to be present. See *dh*(4), *dz*(4), and *dmf*(4) for details.

You should also edit the file /etc/ttytype placing the type of each new terminal there (see *ttytype*(5)).

When the system is running multi-user, all terminals that are listed in /etc/ttys having a 1 as the first character of their line are enabled. If, during normal operations, it is desired to disable a terminal line, you can edit the file /etc/ttys and change the first character of the corresponding line to be a 0 and then send a hangup signal to the *init* process, by doing

    # kill −1 1

Terminals can similarly be enabled by changing the first character of a line from a 0 to a 1 and sending a hangup signal to *init*.

Note that several programs, /usr/src/etc/init.c and /usr/src/etc/comsat.c in particular, will have to be recompiled if there are to be more than 100 terminals. Also note that if a special file is inaccessible when *init* tries to create a process for it, init will print a message on the console and try to reopen the terminal every minute, reprinting the warning message every 10 minutes.

Finally note that you should change the names of any dialup terminals to ttyd? where ? is in [0-9a-f], as some programs use this property of the names to determine if a terminal is a dialup. Shell commands to do this should be put in the /dev/MAKEDEV.local script.

While it is possible to use truly arbitrary strings for terminal names, the accounting and noticeably the *ps*(1) command make good use of the convention that tty names (by default, and also after dialups are named as suggested above) are distinct in the last 2 characters. Change this and you may be sorry later, as the heuristic *ps*(1) uses based on these conventions will then break down and ps will run MUCH slower.

4.6. Adding users

New users can be added to the system by adding a line to the password file /etc/passwd. The procedure for adding a new user is described in *adduser*(8).

You should add accounts for the initial user community, giving each a directory and a password, and putting users who will wish to share software in the same groups.

A number of guest accounts have been provided on the distribution system; these accounts are for people at Berkeley, DEC and at Bell Laboratories who have done major work on UNIX in the past. You can delete these accounts, or leave them on the system if you expect that these people would have occasion to login as guests on your system.

4.7. Site tailoring

All programs which require the site's name, or some similar characteristic, obtain the information through system calls or from files located in /etc. Aside from parts of the system related to the network, to tailor the system to your site you must simply select a site name, then edit the file

    /etc/rc.local

The first line in /etc/rc.local,

    /bin/hostname *mysitename*

defines the value returned by the *gethostname*(2) system call. Programs such as *getty*(8),

September 22, 1983

mail (1), *wall* (1), *uucp* (1), and *who* (1) use this system call so that the binary images are site independent.

4.8. Setting up the line printer system

The line printer system consists of at least the following files and commands:

/usr/ucb/lpq	spooling queue examination program
/usr/ucb/lprm	program to delete jobs from a queue
/usr/ucb/lpr	program to enter a job in a printer queue
/etc/printcap	printer configuration and capability data base
/usr/lib/lpd	line printer daemon, scans spooling queues
/etc/lpc	line printer control program

The file /etc/printcap is a master data base describing line printers directly attached to a machine and, also, printers accessible across a network. The manual page *printcap* (5) describes the format of this data base and also indicates the default values for such things as the directory in which spooling is performed. The line printer system handles multiple printers, multiple spooling queues, local and remote printers, and also printers attached via serial lines which require line initialization such as the baud rate. Raster output devices such as a Varian or Versatec, and laser printers such as an Imagen, are also supported by the line printer system.

Remote spooling via the network is handled with two spooling queues, one on the local machine and one on the remote machine. When a remote printer job is initiated with *lpr*, the job is queued locally and a daemon process created to oversee the transfer of the job to the remote machine. If the destination machine is unreachable, the job will remain queued until it is possible to transfer the files to the spooling queue on the remote machine. The *lpq* program shows the contents of spool queues on both the local and remote machines.

To configure your line printers, consult the printcap manual page and the accompanying document, "4.2BSD Line Printer Spooler Manual". A call to the *lpd* program should be present in /etc/rc.

4.9. Setting up the mail system

The mail system consists of the following commands:

/bin/mail	old standard mail program (from 32/V)
/usr/ucb/mail	UCB mail program, described in mail(1)
/usr/lib/sendmail	mail routing program
/usr/spool/mail	mail spooling directory
/usr/spool/secretmail	secure mail directory
/usr/bin/xsend	secure mail sender
/usr/bin/xget	secure mail receiver
/usr/lib/aliases	mail forwarding information
/usr/ucb/newaliases	command to rebuild binary forwarding database
/usr/ucb/biff	mail notification enabler
/etc/comsat	mail notification daemon
/etc/syslog	error message logger, used by sendmail

Mail is normally sent and received using the *mail* (1) command, which provides a front-end to edit the messages sent and received, and passes the messages to *sendmail* (8) for routing. The routing algorithm uses knowledge of the network name syntax, aliasing and forwarding information, and network topology, as defined in the configuration file /usr/lib/sendmail.cf, to process each piece of mail. Local mail is delivered by giving it to the program /usr/bin/mail which adds it to the mailboxes in the directory /usr/spool/mail/*username*, using a locking protocol to avoid problems with simultaneous updates. After the mail is delivered, the local mail delivery

September 22, 1983

daemon /etc/comsat is notified, which in turn notifies users who have issued a "biff y" command that mail has arrived.

Mail queued in the directory /usr/spool/mail is normally readable only by the recipient. To send mail which is secure against any possible perusal (except by a code-breaker) you should use the secret mail facility, which encrypts the mail so that no one can read it.

To setup the mail facility you should read the instructions in the file READ_ME in the directory /usr/src/usr.lib/sendmail and then adjust the necessary configuration files. You should also set up the file /usr/lib/aliases for your installation, creating mail groups as appropriate. Documents describing *sendmail*'s operation and installation are also included in the distribution.

4.9.1. Setting up a uucp connection

The version of *uucp* included in 4.2BSD is an enhanced version of that originally distributed with 32/V*. The enhancements include:

- support for many auto call units other than the DEC DN11,
- breakup of the spooling area into multiple subdirectories,
- addition of an *L.cmds* file to control the set of commands which may be executed by a remote site,
- enhanced "expect-send" sequence capabilities when logging in to a remote site,
- new commands to be used in polling sites and obtaining snap shots of *uucp* activity.

This section gives a brief overview of *uucp* and points out the most important steps in its installation.

To connect two UNIX machines with a *uucp* network link using modems, one site must have an automatic call unit and the other must have a dialup port. It is better if both sites have both.

You should first read the paper in volume 2B of the Unix Programmers Manual: "Uucp Implementation Description". It describes in detail the file formats and conventions, and will give you a little context. In addition, the document setup.tblms, located in the directory /usr/src/usr.bin/uucp/UUAIDS, may be of use in tailoring the software to your needs.

The *uucp* support is located in three major directories: /usr/bin, /usr/lib/uucp, and /usr/spool/uucp. User commands are kept in /usr/bin, operational commands in /usr/lib/uucp, and /usr/spool/uucp is used as a spooling area. The commands in /usr/bin are:

/usr/bin/uucp	file-copy command
/usr/bin/uux	remote execution command
/usr/bin/uusend	binary file transfer using mail
/usr/bin/uuencode	binary file encoder (for *uusend*)
/usr/bin/uudecode	binary file decoder (for *uusend*)
/usr/bin/uulog	scans session log files
/usr/bin/uusnap	gives a snap-shot of *uucp* activity
/usr/bin/uupoll	polls remote system until an answer is received

The important files and commands in /usr/lib/uucp are:

---

* The *uucp* included in this distribution is the result of work by many people; we gratefully acknowledge their contributions, but refrain from mentioning names in the interest of keeping this document current.

September 22, 1983

/usr/lib/uucp/L-devices	list of dialers and hardwired lines
/usr/lib/uucp/L-dialcodes	dialcode abbreviations
/usr/lib/uucp/L.cmds	commands remote sites may execute
/usr/lib/uucp/L.sys	systems to communicate with, how to connect, and when
/usr/lib/uucp/SEQF	sequence numbering control file
/usr/lib/uucp/USERFILE	remote site pathname access specifications
/usr/lib/uucp/uuclean	cleans up garbage files in spool area
/usr/lib/uucp/uucico	*uucp* protocol daemon
/usr/lib/uucp/uuxqt	*uucp* remote execution server

while the spooling area contains the following important files and directories:

/usr/spool/uucp/C.	directory for command, "C." files
/usr/spool/uucp/D.	directory for data, "D.", files
/usr/spool/uucp/X.	directory for command execution, "X.", files
/usr/spool/uucp/D.*machine*	directory for local "D." files
/usr/spool/uucp/D.*machine*X	directory for local "X." files
/usr/spool/uucp/TM.	directory for temporary, "TM.", files
/usr/spool/uucp/LOGFILE	log file of *uucp* activity
/usr/spool/uucp/SYSLOG	log file of *uucp* file transfers

To install *uucp* on your system, start by selecting a site name (less than 8 characters). A *uucp* account must be created in the password file and a password set up. Then, create the appropriate spooling directories with mode 755 and owned by user *uucp*, group *daemon*.

If you have an auto-call unit, the L.sys, L-dialcodes, and L-devices files should be created. The L.sys file should contain the phone numbers and login sequences required to establish a connection with a *uucp* daemon on another machine. For example, our L.sys file looks something like:

```
adiron Any ACU 1200 out0123456789- ogin-EOT-ogin uucp
cbosg Never Slave 300
cbosgd Never Slave 300
chico Never Slave 1200 out2010123456
```

The first field is the name of a site, the second indicates when the machine may be called, the third field specifies how the host is connected (through an ACU, a hardwired line, etc.), then comes the phone number to use in connecting through an auto-call unit, and finally a login sequence. The phone number may contain common abbreviations which are defined in the L-dialcodes file. The device specification should refer to devices specified in the L-devices file. Indicating only ACU causes the *uucp* daemon, *uucico*, to search for any available auto-call unit in L-devices. Our L-dialcodes file is of the form:

```
ucb 2
out 9%
```

while our L-devices file is:

```
ACU cul0 unused 1200 ventel
```

Refer to the README file in the *uucp* source directory for more information about installation.

As *uucp* operates it creates (and removes) many small files in the directories underneath /usr/spool/uucp. Sometimes files are left undeleted; these are most easily purged with the *uuclean* program. The log files can grow without bound unless trimmed back; *uulog* is used to maintain these files. Many useful aids in maintaining your *uucp* installation are included in a subdirectory UUAIDS beneath /usr/src/usr.bin/uucp. Peruse this directory and read the "setup" instructions also located there.

September 22, 1983

5. NETWORK SETUP

4.2BSD provides support for the DARPA standard Internet protocols IP, ICMP, TCP, and UDP. These protocols may be used on top of a variety of hardware devices ranging from the IMP's used in the ARPANET to local area network controllers for the Ethernet. Network services are split between the kernel (communication protocols) and user programs (user services such as TELNET and FTP). This section describes how to configure your system to use the networking support.

5.1. System configuration

To configure the kernel to include the Internet communication protocols, define the INET option and include the pseudo-devices "inet", "pty", and "loop" in your machine's configuration file. The "pty" pseudo-device forces the pseudo terminal device driver to be configured into the system, see *pty*(4), while the "loop" pseudo-device forces inclusion of the software loopback interface driver. The loop driver is used in network testing and also by the mail system.

If you are planning to use the network facilities on a 10Mb/s Ethernet, the pseudo-device "ether" should also be included in the configuration; this forces inclusion of the Address Resolution Protocol module used in mapping between 48-bit Ethernet and 32-bit Internet addresses. Also, if you have an imp, you will need to include the pseudo-device "imp."

Before configuring the appropriate networking hardware, you should consult the manual pages in section 4 of the programmer's manual. The following table lists the devices for which software support exists.

Device name	Manufacturer and product
acc	ACC LH/DH interface to IMP
css	DEC IMP-11A interface to IMP
dmc	DEC DMC-11 (also works with DMR-11)
ec	3Com 10Mb/s Ethernet
en	Xerox 3Mb/s prototype Ethernet (not a product)
hy	NSC Hyperchannel, w/ DR-11B and PI-13 interfaces
il	Interlan 10Mb/s Ethernet
pcl	DEC PCL-11
un	Ungermann-Bass network w/ DR-11W interface
vv	Proteon ring network (V2LNI)

All network interface drivers require some or all of their host address be defined at boot time. This is accomplished with *ifconfig*(8C) commands included in the /etc/rc.local file. Interfaces which are able to dynamically deduce the host part of an address, but not the network number, take the network number from the address specified with *ifconfig*. Hosts which use a more complex address mapping scheme, such as the Address Resolution Protocol, *arp*(4), require the full address. The manual page for each network interface describes the method used to establish a host's address. *Ifconfig*(8) can also be used to set options for the interface at boot time. These options include disabling the use of the Address Resolution Protocol and/or the use of trailer encapsulation; this is useful if a network is shared with hosts running software which is unable to perform these functions. Options are set independently for each interface, and apply to all packets sent using that interface. An alternative approach to ARP is to divide the address range, using ARP only for those addresses below the cutoff and using another mapping above this constant address; see the source (/sys/netinet/if_ether.c) for more information.

In order to use the pseudo terminals just configured, device entries must be created in the /dev directory. To create 16 pseudo terminals (plenty, unless you have a heavy network load)

September 22, 1983

perform the following commands.

    # cd /dev
    # MAKEDEV pty0

More pseudo terminals may be made by specifying *pty1*, *pty2*, etc. The kernel normally includes support for 32 pseudo terminals unless the configuration file specifies a different number. Each pseudo terminal actually consists of two files in /dev: a master and a slave. The master pseudo terminal file is named /dev/pty?, while the slave side is /dev/ttyp?. Pseudo terminals are also used by the *script*(1) program. In addition to creating the pseudo terminals, be sure to install them in the *letc/ttys* file (with a '0' in the first column so no *getty* is started), and in the *letc/ttytype* file (with type "network").

When configuring multiple networks some thought must be given to the ordering of the devices in the configuration file. The first network interface configured in the system is used as the default network when the system is forced to assign a local address to a socket. This means that your most widely known network should always be placed first in the configuration file. For example, hosts attached to both the ARPANET and our local area network have devices configured in the order show below.

    device   acc0   at uba? csr 0167600 vector accrint accxint
    device   en0   at uba? csr 0161000 vector enxint enrint encollide

5.2. Network data bases

A number of data files are used by the network library routines and server programs. Most of these files are host independent and updated only rarely.

File	Manual reference	Use
/etc/hosts	*hosts*(5)	host names
/etc/networks	*networks*(5)	network names
/etc/services	*services*(5)	list of known services
/etc/protocols	*protocols*(5)	protocol names
/etc/hosts.equiv	*rshd*(8C)	list of "trusted" hosts
/etc/rc.local	*rc*(8)	command script for starting servers
/etc/ftpusers	*ftpd*(8C)	list of "unwelcome" ftp users

The files distributed are set up for ARPANET or other Internet hosts. Local networks and hosts should be added to describe the local configuration; the Berkeley entries may serve as examples (see also the next section). Network numbers will have to be chosen for each ethernet. For sites not connected to the Internet, these can be chosen more or less arbitrarily, otherwise the normal channels should be used for allocation of network numbers.

5.2.1. Regenerating /etc/hosts and /etc/networks

The host and network name data bases are normally derived from a file retrieved from the Internet Network Information Center at SRI. To do this you should use the program /etc/gettable to retrieve the NIC host data base, and the program /etc/htable to convert it to the format used by the libraries.

```
# cd /usr/src/ucb/netser/htable
# /etc/gettable sri-nic
Connection to sri-nic opened.
Host table received.
Connection to sri-nic closed.
# /etc/htable hosts.txt
Warning, no localgateways file.
#
```

The *htable* program generates two files of interest in the local directory: *hosts* and *networks*. If a file "localhosts" is present in the working directory its contents are first copied to the output file. Similarly, a "localnetworks" file may be prepended to the output created by *htable*. It is usually wise to run *diff*(1) on the new host and network data bases before installing them in /etc.

5.2.2. /etc/hosts.equiv

The remote login and shell servers use an authentication scheme based on trusted hosts. The hosts.equiv file contains a list of hosts which are considered trusted and/or, under a single administrative control. When a user contacts a remote login or shell server requesting service, the client process passes the user's name and the official name of the host on which the client is located. In the simple case, if the hosts's name is located in hosts.equiv and the user has an account on the server's machine, then service is rendered (i.e. the user is allowed to log in, or the command is executed). Users may constrain this "equivalence" of machines by installing a .rhosts file in their login directory. The root login is handled specially, bypassing the hosts.equiv file, and using only the /.rhosts file.

Thus, to create a class of equivalent machines, the hosts.equiv file should contain the *official* names for those machines. For example, most machines on our major local network are considered trusted, so the hosts.equiv file is of the form:

```
ucbarpa
ucbcalder
ucbdali
ucbernie
ucbkim
ucbmatisse
ucbmonet
ucbvax
ucbmiro
ucbdegas
```

5.2.3. /etc/rc.local

Most network servers are automatically started up at boot time by the command file /etc/rc (if they are installed in their presumed locations). These include the following:

/etc/rshd	shell server
/etc/rexecd	exec server
/etc/rlogind	login server
/etc/rwhod	system status daemon

To have other network servers started up as well, commands of the following sort should be placed in the site dependent file /etc/rc.local.

September 22, 1983

```
if [ -f /etc/telnetd ]; then
        /etc/telnetd & echo -n ' telnetd'              >/dev/console
fi
```

The following servers are included with the system and should be installed in /etc/rc.local as the need arises.

/etc/telnetd	TELNET server
/etc/ftpd	FTP server
/etc/tftpd	TFTP server
/etc/syslog	error logging server
/etc/sendmail	SMTP server
/etc/courierd	Courier remote procedure call server
/etc/routed	routing table management daemon

Consult the manual pages and accompanying documentation (particularly for sendmail) for details about their operation.

5.2.4. /etc/ftpusers

The FTP server included in the system provides support for an anonymous FTP account. Due to the inherent security problems with such a facility you should read this section carefully if you consider providing such a service.

An anonymous account is enabled by creating a user *ftp*. When a client uses the anonymous account a *chroot*(2) system call is performed by the server to restrict the client from moving outside that part of the file system where the user ftp home directory is located. Because a chroot call is used, certain programs and files must be supplied the server process for it to execute properly. Further, one must be sure that all directories and executable images are unwritable. The following directory setup is recommended.

```
# cd ~ftp
# chmod 555 .; chown ftp .; chgrp ftp .
# mkdir bin etc pub
# chown root bin etc
# chmod 555 bin etc
# chown ftp pub
# chmod 777 pub
# cd bin
# cp /bin/sh /bin/ls .
# chmod 111 sh ls
# cd ../etc
# cp /etc/passwd /etc/group .
# chmod 444 passwd group
```

When local users wish to place files in the anonymous area, they must be placed in a subdirectory. In the setup here, the directory ~ftp/pub is used.

Aside from the problems of directory modes and such, the ftp server may provide a loophole for interlopers if certain user accounts are allowed. The file /etc/ftpusers is checked on each connection. If the requested user name is located in the file, the request for service is denied. This file normally has the following names on our systems.

```
uucp
root
```

September 22, 1983

5.3. Routing and gateways/bridges

If your environment allows access to networks not directly attached to your host you will need to set up routing information to allow packets to be properly routed. Two schemes are supported by the system. The first scheme employs the routing table management daemon /etc/routed to maintain the system routing tables. The routing daemon uses a variant of the Xerox Routing Information Protocol to maintain up to date routing tables in a cluster of local area networks. By using the /etc/gateways file created by /etc/htable, the routing daemon can also be used to initialize static routes to distant networks. When the routing daemon is started up (usually from /etc/rc.local) it reads /etc/gateways and installs those routes defined there, then broadcasts on each local network to which the host is attached to find other instances of the routing daemon. If any responses are received, the routing daemons cooperate in maintaining a globally consistent view of routing in the local environment. This view can be extended to include remote sites also running the routing daemon by setting up suitable entries in /etc/gateways; consult *routed* (8C) for a more thorough discussion.

The second approach is to define a wildcard route to a smart gateway and depend on the gateway to provide ICMP routing redirect information to dynamically create a routing data base. This is done by adding an entry of the form

        /etc/route add 0 *smart-gateway* 1

to /etc/rc.local; see *route* (8C) for more information. The wildcard route, indicated by a 0 valued destination, will be used by the system as a "last resort" in routing packets to their destination. Assuming the gateway to which packets are directed is able to generate the proper routing redirect messages, the system will then add routing table entries based on the information supplied. This approach has certain advantages over the routing daemon, but is unsuitable in an environment where their are only bridges (i.e. pseudo gateways which, for instance, do not generate routing redirect messages). Further, if the smart gateway goes down there is no alternative, save manual alteration of the routing table entry, to maintaining service.

The system always listens, and processes, routing table redirect information, so it is possible to combine both the above facilities. For example, the routing table management process might be used to maintain up to date information about routes to geographically local networks, while employing the wildcard routing techniques for "distant" networks. The *netstat* (1) program may be used to display routing table contents as well as various routing oriented statistics. For example,

        # netstat −r

will display the contents of the routing tables, while

        # netstat −r −s

will show the number of routing table entries dynamically created as a result of routing redirect messages, etc.

September 22, 1983

6. SYSTEM OPERATION

This section describes procedures used to operate a VAX UNIX system. Procedures described here are used periodically, to reboot the system, analyze error messages from devices, do disk backups, monitor system performance, recompile system software and control local changes.

6.1. Bootstrap and shutdown procedures

In a normal reboot, the system checks the disks and comes up multi-user without intervention at the console. Such a reboot can be stopped (after it prints the date) with a ^C (interrupt). This will leave the system in single-user mode, with only the console terminal active.

If booting from the console command level is needed, then the command

>>> B

will boot from the default device. On an 11/780 (11/730) the default device is determined by a "DEPOSIT" command stored on the floppy (cassette) in the file "DEFBOO.CMD"; on an 11/750 the default device is determined by the setting of a switch on the front panel.

You can boot a system up single user on a 780 or 730 by doing

>>> B *XX*S

where *XX* is one of HP, HK, UP, RA, or RB for a 730. The corresponding command on an 11/750 is

>>> B/1

For second vendor storage modules on the UNIBUS or MASSBUS of an 11/750 you will need to have a boot prom. Most vendors will sell you such proms for their controllers; contact your vendor if you don't have one.

Other possibilities are:

>>> B ANY

or, on a 750

>>> B/3

These commands boot and ask for the name of the system to be booted. They can be used after building a new test system to give the boot program the name of the test version of the system.

To bring the system up to a multi-user configuration from the single-user status after, e.g., a "B HPS" on an 11/780, "B RBS" on a 730, or a "B/1" on an 11/750 all you have to do is hit ^D on the console. The system will then execute /etc/rc, a multi-user restart script (and /etc/rc.local), and come up on the terminals listed as active in the file /etc/ttys. See *init*(8) and *ttys*(5). Note, however, that this does not cause a file system check to be performed. Unless the system was taken down cleanly, you should run "fsck −p" or force a reboot with *reboot*(8) to have the disks checked.

To take the system down to a single user state you can use

kill 1

or use the *shutdown*(8) command (which is much more polite, if there are other users logged in.) when you are up multi-user. Either command will kill all processes and give you a shell on the console, as if you had just booted. File systems remain mounted after the system is taken single-user. If you wish to come up multi-user again, you should do this by:

```
# cd /
# /etc/umount -a
# ^D
```

Each system shutdown, crash, processor halt and reboot is recorded in the file /usr/adm/shutdownlog with the cause.

6.2. Device errors and diagnostics

When errors occur on peripherals or in the system, the system prints a warning diagnostic on the console. These messages are collected regularly and written into a system error log file /usr/adm/messages.

Error messages printed by the devices in the system are described with the drivers for the devices in section 4 of the programmer's manual. If errors occur indicating hardware problems, you should contact your hardware support group or field service. It is a good idea to examine the error log file regularly (e.g. with "tail −r /usr/adm/messages").

6.3. File system checks, backups and disaster recovery

Periodically (say every week or so in the absence of any problems) and always (usually automatically) after a crash, all the file systems should be checked for consistency by *fsck* (1). The procedures of *reboot* (8) should be used to get the system to a state where a file system check can be performed manually or automatically.

Dumping of the file systems should be done regularly, since once the system is going it is easy to become complacent. Complete and incremental dumps are easily done with *dump* (8). You should arrange to do a towers-of-hanoi dump sequence; we tune ours so that almost all files are dumped on two tapes and kept for at least a week in most every case. We take full dumps every month (and keep these indefinitely). Operators can execute "dump w" at login that will tell them what needs to be dumped (based on the /etc/fstab information). Be sure to create a group **operator** in the file /etc/group so that dump can notify logged-in operators when it needs help.

More precisely, we have three sets of dump tapes: 10 daily tapes, 5 weekly sets of 2 tapes, and fresh sets of three tapes monthly. We do daily dumps circularly on the daily tapes with sequence '3 2 5 4 7 6 9 8 9 9 9 ...'. Each weekly is a level 1 and the daily dump sequence level restarts after each weekly dump. Full dumps are level 0 and the daily sequence restarts after each full dump also.

Thus a typical dump sequence would be:

tape name	level number	date	opr	size
FULL	0	Nov 24, 1979	jkf	137K
D1	3	Nov 28, 1979	jkf	29K
D2	2	Nov 29, 1979	rrh	34K
D3	5	Nov 30, 1979	rrh	19K
D4	4	Dec 1, 1979	rrh	22K
W1	1	Dec 2, 1979	etc	40K
D5	3	Dec 4, 1979	rrh	15K
D6	2	Dec 5, 1979	jkf	25K
D7	5	Dec 6, 1979	jkf	15K
D8	4	Dec 7, 1979	rrh	19K
W2	1	Dec 9, 1979	etc	118K
D9	3	Dec 11, 1979	rrh	15K
D10	2	Dec 12, 1979	rrh	26K
D1	5	Dec 15, 1979	rrh	14K
W3	1	Dec 17, 1979	etc	71K
D2	3	Dec 18, 1979	etc	13K

September 22, 1983

FULL   0   Dec 22, 1979   etc   135K

We do weekly's often enough that daily's always fit on one tape and never get to the sequence of 9's in the daily level numbers.

Dumping of files by name is best done by *tar* (1) but the amount of data that can be moved in this way is limited to a single tape. Finally if there are enough drives entire disks can be copied with *dd* (1) using the raw special files and an appropriate blocking factor; the number of sectors per track is usually a good value to use, consult /etc/disktab.

It is desirable that full dumps of the root file system be made regularly. This is especially true when only one disk is available. Then, if the root file system is damaged by a hardware or software failure, you can rebuild a workable disk doing a restore in the same way that the initial root file system was created.

Exhaustion of user-file space is certain to occur now and then; disk quotas may be imposed, or if you prefer a less facist approach, try using the programs *du* (1), *df*(1), *quot*(8), combined with threatening messages of the day, and personal letters.

6.4. Moving filesystem data

If you have the equipment, the best way to move a file system is to dump it to magtape using *dump* (8), use *newfs* (8) to create the new file system, and restore the tape, using *restore* (8). If for some reason you don't want to use magtape, dump accepts an argument telling where to put the dump; you might use another disk. The restore program uses an "inplace" algorithm which allows file system dumps to be restored without concern for the original size of the file system. Further, portions of a file system may be selectively restored in a manner similar to the tape archive program.

If you have to merge a file system into another, existing one, the best bet is to use *tar* (1). If you must shrink a file system, the best bet is to dump the original and restore it onto the new file system. If you are playing with the root file system and only have one drive, the procedure is more complicated. What you do is the following:

1.   GET A SECOND PACK!!!!

2.   Dump the root file system to tape using *dump*(8).

3.   Bring the system down and mount the new pack.

4.   Load the distribution tape and install the new root file system as you did when first installing the system.

5.   Boot normally using the newly created disk file system.

Note that if you change the disk partition tables or add new disk drivers they should also be added to the standalone system in /sys/stand and the default disk partition tables in /etc/disktab should be modified.

6.5. Monitoring System Performance

The *vmstat* program provided with the system is designed to be an aid to monitoring systemwide activity. Together with the *ps* (1) command (as in "ps av"), it can be used to investigate systemwide virtual memory activity. By running *vmstat* when the system is active you can judge the system activity in several dimensions: job distribution, virtual memory load, paging and swapping activity, disk and cpu utilization. Ideally, there should be few blocked (b) jobs, there should be little paging or swapping activity, there should be available bandwidth on the disk devices (most single arms peak out at 30-35 tps in practice), and the user cpu utilization (us) should be high (above 60%).

If the system is busy, then the count of active jobs may be large, and several of these jobs may often be blocked (b). If the virtual memory is active, then the paging demon will be running (sr will be non-zero). It is healthy for the paging demon to free pages when the virtual memory gets active; it is triggered by the amount of free memory dropping below a threshold

and increases its pace as free memory goes to zero.

If you run *vmstat* when the system is busy (a "vmstat 1" gives all the numbers computed by the system), you can find imbalances by noting abnormal job distributions. If many processes are blocked (b), then the disk subsystem is overloaded or imbalanced. If you have a several non-dma devices or open teletype lines that are "ringing", or user programs that are doing high-speed non-buffered input/output, then the system time may go high (60-70% or higher). It is often possible to pin down the cause of high system time by looking to see if there is excessive context switching (cs), interrupt activity (in) or system call activity (sy). Cumulatively on one of our large machines we average about 60 context switches and interrupts per second and about 90 system calls per second.

If the system is heavily loaded, or if you have little memory for your load (1M is little in most any case), then the system may be forced to swap. This is likely to be accompanied by a noticeable reduction in system performance and pregnant pauses when interactive jobs such as editors swap out. If you expect to be in a memory-poor environment for an extended period you might consider administratively limiting system load.

6.6. Recompiling and reinstalling system software

It is easy to regenerate the system, and it is a good idea to try rebuilding pieces of the system to build confidence in the procedures. The system consists of two major parts: the kernel itself (/sys) and the user programs (/usr/src and subdirectories). The major part of this is /usr/src.

The three major libraries are the C library in /usr/src/lib/libc and the FORTRAN libraries /usr/src/usr.lib/libI77 and /usr/src/usr.lib/libF77. In each case the library is remade by changing into the corresponding directory and doing

    # make

and then installed by

    # make install

Similar to the system,

    # make clean

cleans up.

The source for all other libraries is kept in subdirectories of /usr/src/usr.lib; each has a makefile and can be recompiled by the above recipe.

If you look at /usr/src/Makefile, you will see that you can recompile the entire system source with one command. To recompile a specific program, find out where the source resides with the *whereis*(1) command, then change to that directory and remake it with the makefile present in the directory. For instance, to recompile "date", all one has to do is

    # whereis date
    date: /usr/src/bin/date.c /bin/date /usr/man/man1/date.1
    # cd /usr/src/bin
    # make date

this will create an unstripped version of the binary of "date" in the current directory. To install the binary image, use the install command as in

    # install −s date /bin/date

The −s option will insure the installed version of date has its symbol table stripped. The install command should be used instead of mv or cp as it understands how to install programs even when the program is currently in use.

September 22, 1983

If you wish to recompile and install all programs in a particular target area you can override the default target by doing:

```
# make
# make DESTDIR=pathname install
```

To regenerate all the system source you can do

```
# cd /usr/src
# make
```

If you modify the C library, say to change a system call, and want to rebuild and install everything from scratch you have to be a little careful. You must insure the libraries are installed before the remainder of the source, otherwise the loaded images will not contain the new routine from the library. The following steps are recommended.

```
# cd /usr/src
# cd lib; make install
# cd ..
# make usr.lib
# cd usr.lib; make install
# cd ..
# make bin etc usr.bin ucb games local
# for i in bin etc usr.bin ucb games local; do (cd $i; make install); done
```

This will take about 12 hours on a reasonably configured 11/750.

6.7. Making local modifications

To keep track of changes to system source we migrate changed versions of commands in /usr/src/bin, /usr/src/usr.bin, and /usr/src/ucb in through the directory /usr/src/new and out of the original directory into /usr/src/old for a time before removing them. Locally written commands that aren't distributed are kept in /usr/src/local and their binaries are kept in /usr/local. This allows /usr/bin, /usr/ucb, and /bin to correspond to the distribution tape (and to the manuals that people can buy). People wishing to use /usr/local commands are made aware that they aren't in the base manual. As manual updates incorporate these commands they are moved to /usr/ucb.

A directory /usr/junk to throw garbage into, as well as binary directories /usr/old and /usr/new are useful. The man command supports manual directories such as /usr/man/manj for junk and /usr/man/manl for local to make this or something similar practical.

6.8. Accounting

UNIX optionally records two kinds of accounting information: connect time accounting and process resource accounting. The connect time accounting information is stored in the file /usr/adm/wtmp, which is summarized by the program ac(8). The process time accounting information is stored in the file /usr/adm/acct, and analyzed and summarized by the program sa(8).

If you need to implement recharge for computing time, you can implement procedures based on the information provided by these commands. A convenient way to do this is to give commands to the clock daemon /etc/cron to be executed every day at a specified time. This is done by adding lines to /usr/adm/crontab; see cron(8) for details.

6.9. Resource control

Resource control in the current version of UNIX is fairly elaborate compared to most UNIX systems. The disc quota facilities developed at the University of Melbourne have been incorporated in the system and allow control over the number of files and amount of disc space

each user may use on each file system. In addition, the resources consumed by any single process can be limited by the mechanisms of *setrlimit* (2). As distributed, the latter mechanism is voluntary, though sites may choose to modify the login mechanism to impose limits not covered with disc quotas.

To utilize the disc quota facilities, the system must be configured with "options QUOTA". File systems may then be placed under the quota mechanism by creating a null file *quotas* at the root of the file system, running *quotacheck* (8), and modifying /etc/fstab to indicate the file system is read-write with disc quotas (a "rq" type field). The *quotaon* (8) program may then be run to enable quotas.

Individual quotas are applied by using the quota editor *edquota* (8). Users may view their quotas (but not those of other users) with the *quota* (1) program. The *repquota* (8) program may be used to summarize the quotas and current space usage on a particular file system or file systems.

Quotas are enforced with *soft* and *hard* limits. When a user first reaches a soft limit on a resource, a message is generated on his/her terminal. If the user fails to lower the resource usage below the soft limit the next time they log in to the system the *login* program will generate a warning about excessive usage. Should three login sessions go by with the soft limit breached the system then treats the soft limit as a *hard* limit and disallows any allocations until enough space is reclaimed to bring the user back below the soft limit. Hard limits are enforced strictly resulting in errors when a user tries to create or write a file. Each time a hard limit is exceeded the system will generate a message on the user's terminal.

Consult the auxiliary document, "Disc Quotas in a UNIX Environment" and the appropriate manual entries for more information.

6.10. Network troubleshooting

If you have anything more than a trivial network configuration, from time to time you are bound to run into problems. Before blaming the software, first check your network connections. On networks such as the Ethernet a loose cable tap or misplaced power cable can result in severely deteriorated service. The *netstat* (1) program may be of aid in tracking down hardware malfunctions. In particular, look at the −i and −s options in the manual page.

Should you believe a communication protocol problem exists, consult the protocol specifications and attempt to isolate the problem in a packet trace. The SO_DEBUG option may be supplied before establishing a connection on a socket, in which case the system will trace all traffic and internal actions (such as timers expiring) in a circular trace buffer. This buffer may then be printed out with the *trpt* (8C) program. Most all the servers distributed with the system accept a −d option forcing all sockets to be created with debugging turned on. Consult the appropriate manual pages for more information.

6.11. Files which need periodic attention

We conclude the discussion of system operations by listing the files that require periodic attention or are system specific

/etc/fstab	how disk partitions are used
/etc/disktab	disk partition sizes
/etc/printcap	printer data base
/etc/gettytab	terminal type definitions
/etc/remote	names and phone numbers of remote machines for *tip*(1)
/etc/group	group memberships
/etc/motd	message of the day
/etc/passwd	password file; each account has a line
/etc/rc.local	local system restart script; runs reboot; starts daemons
/etc/hosts	host name data base
/etc/networks	network name data base

September 22, 1983

/etc/services	network services data base
/etc/hosts.equiv	hosts under same administrative control
/etc/securetty	restricted list of ttys where root can log in
/etc/ttys	enables/disables ports
/etc/ttytype	terminal types connected to ports
/usr/lib/crontab	commands that are run periodically
/usr/lib/aliases	mail forwarding and distribution groups
/usr/adm/acct	raw process account data
/usr/adm/messages	system error log
/usr/adm/shutdownlog	log of system reboots
/usr/adm/wtmp	login session accounting

APPENDIX A — BOOTSTRAP DETAILS

This appendix contains pertinent files and numbers regarding the bootstrapping procedure for 4.2BSD. You should never have to look at this appendix. However, if there are problems in installing the distribution on your machine, the material contained here may prove useful.

Contents of the distribution tapes

The distribution normally consists of two 1600bpi 2400' magnetic tapes. The first tape contains the following files on it. All tape files are blocked in 10 kilobytes records, except for the first file on the first tape which has 512 byte records.

Tape file	Records*	Contents
one	194	8 bootstrap monitor programs and a
		tp(1) file containing *boot*, *format*, and *copy*
two	205	"mini root" file system
three	380	*dump*(8) of distribution root file system
four	440	*tar*(1) image of /sys, including GENERIC system
five	2111	*tar*(1) image of binaries and libraries in /usr
six	576	*tar*(1) image of /usr/lib/vfont

The second tape contains the following files.

Tape file	# Records	Contents
one	2100	*tar*(1) image of /usr/src
two	973	*tar*(1) image of user contributed software
three	420	*tar*(1) image of /usr/ingres

The distribution tape is made with the shell scripts located in the directory /sys/dist. To construct a distribution tape one must first build a mini root file system with the *buildmini* shell script.

```
#! /bin/sh
#     @(#)buildmini 4.4   7/9/83
#
miniroot=hp0g
minitype=rm80
#
date
umount /dev/${miniroot}
newfs -s 4096 ${miniroot} ${minitype}
fsck /dev/r${miniroot}
mount /dev/${miniroot} /mnt
cd /mnt; sh /sys/dist/get
cd /sys/dist; sync
umount /dev/${miniroot}
fsck /dev/${miniroot}
date
```

The *buildmini* script uses the *get* script to construct the actual file system.

---

* The number of records in each tape file may not be precisely that shown in this table; these values reflect the contents of the distribution tape at the time this document was written.

September 22, 1983

```
#! /bin/sh
#       @(#)get   4.13   7/19/83
#
# Shell script to build a mini-root file system
# in preparation for building a distribution tape.
# The file system created here is image copied onto
# tape, then image copied onto disk as the "first"
# step in a cold boot of 4.2 systems.
#
DISTROOT=/nbsd
#
if [ 'pwd' = '/' ]
then
      echo You just '(almost)' destroyed the root
      exit
fi
cp $DISTROOT/a/sys/GENERIC/vmunix .
rm -rf bin; mkdir bin
rm -rf etc; mkdir etc
rm -rf a; mkdir a
rm -rf tmp; mkdir tmp
rm -rf usr; mkdir usr usr/mdec
rm -rf sys; mkdir sys sys/floppy sys/cassette
cp $DISTROOT/etc/disktab etc
cp $DISTROOT/etc/newfs etc; strip etc/newfs
cp $DISTROOT/etc/mkfs etc; strip etc/mkfs
cp $DISTROOT/etc/restore etc; strip etc/restore
cp $DISTROOT/etc/init etc; strip etc/init
cp $DISTROOT/etc/mount etc; strip etc/mount
cp $DISTROOT/etc/mknod etc; strip etc/mknod
cp $DISTROOT/etc/fsck etc; strip etc/fsck
cp $DISTROOT/etc/umount etc; strip etc/umount
cp $DISTROOT/etc/arff etc; strip etc/arff
cp $DISTROOT/etc/flcopy etc; strip etc/flcopy
cp $DISTROOT/bin/mt bin; strip bin/mt
cp $DISTROOT/bin/ls bin; strip bin/ls
cp $DISTROOT/bin/sh bin; strip bin/sh
cp $DISTROOT/bin/mv bin; strip bin/mv
cp $DISTROOT/bin/sync bin; strip bin/sync
cp $DISTROOT/bin/cat bin; strip bin/cat
cp $DISTROOT/bin/mkdir bin; strip bin/mkdir
cp $DISTROOT/bin/stty bin; strip bin/stty; ln bin/stty bin/STTY
cp $DISTROOT/bin/echo bin; strip bin/echo
cp $DISTROOT/bin/rm bin; strip bin/rm
cp $DISTROOT/bin/cp bin; strip bin/cp
cp $DISTROOT/bin/expr bin; strip bin/expr
cp $DISTROOT/bin/awk bin; strip bin/awk
cp $DISTROOT/bin/make bin; strip bin/make
cp $DISTROOT/usr/mdec/* usr/mdec
cp $DISTROOT/a/sys/floppy/[Ma-z0-9]* sys/floppy
cp $DISTROOT/a/sys/cassette/[Ma-z0-9]* sys/cassette
cp $DISTROOT/a/sys/stand/boot boot
cp $DISTROOT/.profile .profile
cat >etc/passwd <<EOF
```

```
        root::0:10::/:/bin/sh
        EOF
        cat >etc/group <<EOF
        wheel:*:0:
        staff:*:10:
        EOF
        cat >etc/fstab <<EOF
        /dev/hp0a:/a:xx:1:1
        /dev/up0a:/a:xx:1:1
        /dev/hk0a:/a:xx:1:1
        /dev/ra0a:/a:xx:1:1
        /dev/rb0a:/a:xx:1:1
        EOF
        cat >xtr <<'EOF'
        : ${disk?'Usage: disk=xx0 type=tt tape=yy xtr'}
        : ${type?'Usage: disk=xx0 type=tt tape=yy xtr'}
        : ${tape?'Usage: disk=xx0 type=tt tape=yy xtr'}
        echo 'Build root file system'
        newfs ${disk}a ${type}
        sync
        echo 'Check the file system'
        fsck /dev/r${disk}a
        mount /dev/${disk}a /a
        cd /a
        echo 'Rewind tape'
        mt -t /dev/${tape}0 rew
        echo 'Restore the dump image of the root'
        restore rsf 3 /dev/${tape}0
        cd /
        sync
        umount /dev/${disk}a
        sync
        fsck /dev/r${disk}a
        echo 'Root filesystem extracted'
        echo
        echo 'If this is a 780, update floppy'
        echo 'If this is a 730, update the cassette'
        EOF
        chmod +x xtr
        rm -rf dev; mkdir dev
        cp $DISTROOT/sys/dist/MAKEDEV dev
        chmod +x dev/MAKEDEV
        cp /dev/null dev/MAKEDEV.local
        cd dev
        cd ..
        sync
```

The mini root file system must have enough space to hold the files found on a floppy or cassette.

Once a mini root file system is constructed, the *maketape* script is used to make a distribution tape.

```
#! /bin/sh
#     @(#)maketape 4.12  8/4/83
#
miniroot=hp0g
#
trap "rm -f /tmp/tape.$$; exit" 0 1 2 3 13 15
mt rew
date
umount /dev/hp2g /dev/hp2h
umount /dev/hp2a
mount -r /dev/hp2a /nbsd
mount -r /dev/hp2g /nbsd/usr
mount -r /dev/hp2h /nbsd/a
cd /nbsd/tp
tp cmf /tmp/tape.$$ boot copy format
cd /nbsd/sys/mdec
echo "Build 1st level boot block file"
cat tsboot htboot tmboot mtboot utboot noboot noboot /tmp/tape.$$ | \
      dd of=/dev/rmt12 bs=512 conv=sync
cd /nbsd
sync
echo "Add dump of mini-root file system"
dd if=/dev/r${miniroot} of=/dev/rmt12 bs=20b count=205 conv=sync
echo "Add full dump of real file system"
/etc/dump 0uf /dev/rmt12 /nbsd
echo "Add tar image of system sources"
cd /nbsd/a/sys; tar cf /dev/rmt12 .
echo "Add tar image of /usr"
cd /nbsd/usr; tar cf /dev/rmt12 adm bin dict doc games \
      guest hosts include lib local man mdec msgs new \
      old preserve pub spool tmp ucb
echo "Add varian fonts"
cd /usr/lib/vfont; tar cf /dev/rmt12 .
echo "Done, rewinding first tape"
mt rew
echo "Mount second tape and hit return when ready"; read x
echo "Add user source code"
cd /nbsd/usr/src; tar cf /dev/rmt12 .
echo "Add user contributed software"
cd /usr/src/new; tar cf /dev/rmt12 .
echo "Add ingres source"
cd /nbsd/usr/ingres; tar cf /dev/rmt12 .
echo "Done, rewinding second tape"
mt rew
```

Summarizing then, to construct a distribution tape you can use the above scripts and the following commands.

September 22, 1983

buildmini
maketape
...
Done, rewinding first tape
Mount second tape and hit return when ready
(remove the first tape and place a fresh one on the drive)

...
Done, rewinding second tape

Control status register addresses

The distribution uses many standalone device drivers which presume the location of a UNIBUS device's control status register (CSR). The following table summarizes these values.

Device name	Controller	CSR address (octal)
ra	DEC UDA50	0172150
rb	DEC 730 IDC	0175606
rk	DEC RK11	0177440
rl	DEC RL11	0174400
tm	EMULEX TC-11	0172520
ts	DEC TS11	0172520
up	EMULEX SC-21V	0176700
ut	SI 9700	0172440

All MASSBUS controllers are located at standard offsets from the base address of the MASSBUS adapter register bank.

Generic system control status register addresses

The *generic* version of the operating system supplied with the distribution contains the UNIBUS devices indicated below. These devices will be recognized if the appropriate control status registers respond at any of the indicated UNIBUS addresses.

Device name	Controller	CSR addresses (octal)
hk	DEC RK11	0177440
tm	EMULEX TC-11	0172520
ut	SI 9700	0172440
up	EMULEX SC-2 V	0176700, 0174400, 0176300
ra	DEC UDA-50	0172150, 0172550, 0177550
rb	DEC 730 IDC	0175606
rl	DEC RL11	0174400
dn	DEC DN11	0160020
dm	DM11 equivalent	0170500
dh	DH11 equivalent	0160040
dz	DEC DZ11	0160100, 0160110, ... 0160170
ts	DEC TS11	0172520
dmf	DEC DMF32	0160340
lp	DEC LP11	0177514

If devices other than the above are located at any of the addresses indicated, the system may not bootstrap properly.

September 22, 1983

APPENDIX B — LOADING THE TAPE MONITOR

This section indicates how the bootstrap monitor located on the first tape of the distribution tape set may be loaded. This should not be necessary, but has been included as a fallback measure in case it is not possible to read the distributed console medium. **WARNING**: the bootstraps supplied below may not work, in certain instances on an 11/730 because they use a buffered data path for transferring data from tape to memory; consult our group if you are unable to load the monitor on an 11/730.

To load the tape bootstrap monitor, first mount the magnetic tape on drive 0 at load point, making sure that the write ring is not inserted. Temporarily set the reboot switch on an 11/780 or 11/730 to off; on an 11/750 set the power-on action to halt. (In normal operation an 11/780 or 11/730 will have the reboot switch on, and an 11/750 will have the power-on action set to boot/restart.)

If you have an 11/780 give the commands:

>>> HALT
>>> UNJAM

Then, on any machine, give the init command:

>>> I

and then key in at location 200 and execute either the TS, HT, TM, or MT bootstrap that follows, as appropriate. The machine's printouts are shown in boldface, explanatory comments are within ( ). (You can use 'delete' to delete a character and 'control U' to kill the whole line.)

TS bootstrap

```
>>> D/P 200 3AEFD0
>>> D + D05A0000
>>> D + 3BEF
>>> D + 800CA00
>>> D + 32EFC1
>>> D + CA010000
>>> D + EFC10804
>>> D + 24
>>> D + 15508F
>>> D + ABB45B00
>>> D + 2AB9502
>>> D + 8FB0FB18
>>> D + 956B024C
>>> D + FB1802AB
>>> D + 25C8FB0
>>> D + 6B
     (The next two deposits set up the addresses of the UNIBUS)
     (adapter and its memory; the 20006000 here is the address of)
     (the 11/780 uba0 and the 2013E000 the address of the 11/780 umem0)
>>> D + 20006000        (780 uba0)
     (780 uba1: 20008000, 750 uba: F30000, 730 uba: F26000)
>>> D + 2013E000        (780 umem0)
     (780 umem1: 2017E000, 750 umem: FFE000, 730 umem: FFE000)
>>> D + 80000000
>>> D + 254C004
```

September 22, 1983

```
>>> D + 80000
>>> D + 264
>>> D + E
>>> D + C001
>>> D + 2000000
>>> S 200
```

HT bootstrap

```
>>> D/P 200 3EEFD0
>>> D + C55A0000
>>> D + 3BEF
>>> D + 808F00
>>> D + C15B0000
>>> D + C05B5A5B
>>> D + 4008F
>>> D + D05B00
>>> D + 9D004AA
>>> D + C08F326B
>>> D + D424AB14
>>> D + 8FD00CAA
>>> D + 80000000
>>> D + 320800CA
>>> D + AAFE008F
>>> D + 6B39D010
>>> D + 0
```
        (The next two deposits set up the addresses of the MASSBUS)
        (adapter and the drive number for the tape formatter)
        (the 20012000 here is the address of the 11/780 mba1 and the 0)
        (reflects that the formatter is drive 0 on mba1)
```
>>> D + 20012000          (780 mba1) (780 mba0: 20010000, 750 mba0: F28000)
>>> D + 0                 (Formatter unit number in range 0-7)
>>> S 200
>>> S 200
```

TM bootstrap

```
>>> D/P 200 2AEFD0
>>> D + D0510000
>>> D + 2000008F
>>> D + 800C180
>>> D + 804C1D4
>>> D + 1AEFD0
>>> D + C8520000
>>> D + F5508F
>>> D + 8FAE5200
>>> D + 4A20200
>>> D + B006A2B4
>>> D + 2A203
```
        (The following two numbers are uba0 and umem0; see TS above)
        (for some hints on other values if your TM isn't on UBA0 on a 780)
```
>>> D + 20006000          (780 uba0)
>>> D + 2013E000          (780 umem0)
>>> S 200
```

```
>>> S 200
>>> S 200
```

MT bootstrap

```
>>> D/P 200 46EFD0
>>> D + C55A0000
>>> D + 43EF
>>> D + 808F00
>>> D + C15B0000
>>> D + C05B5A5B
>>> D + 4008F
>>> D + 19A5B00
>>> D + 49A04AA
>>> D + AAD408AB
>>> D + 8FD00C
>>> D + CA800000
>>> D + 8F320800
>>> D + 10AAFE00
>>> D + 2008F3C
>>> D + ABD014AB
>>> D + FE15044
>>> D + 399AF850
>>> D + 6B
```
       (The next two deposits set up the addresses of the MASSBUS)
       (adapter and the drive number for the tape formatter)
       (the 20012000 here is the address of the 11/780 mba1 and the 0)
       (reflects that the formatter is drive 0 on mba1)
```
>>> D + 20012000
>>> D + 0
>>> S 200
>>> S 200
>>> S 200
>>> S 200
```

(no toggle-in code exists for the UT device)

If the tape doesn't move the first time you start the bootstrap program with "S 200" you probably have entered the program incorrectly (but also check that the tape is online). Start over and check your typing. For the HT, MT, and TM bootstraps you will not be able to see the tape motion as you advance through the first few blocks as the tape motion is all within the vacuum columns.

Next, deposit in RA the address of the tape MBA/UBA and in RB the address of the device registers or unit number from one of:

```
>>> D/G A 20006000          (for tapes on 780 uba0)
>>> D/G A 20008000          (for tapes on 780 uba1)
>>> D/G A 20012000          (for tapes on 780 mba1)
>>> D/G A 20010000          (for tapes on 780 mba0)
>>> D/G A F30000            (for tapes on 750 uba0)
>>> D/G A F2A000            (for tapes on 750 mba1)
>>> D/G A F28000            (for tapes on 750 mba0)
>>> D/G A F26000            (for tapes on 730 uba0)
```

September 22, 1983

and for register B:

```
>>> D/G B 0              (for tm03/tm78 formatters at mba? drive 0)
>>> D/G B 1              (for tm03/tm78 formatters at mba? drive 1)
>>> D/G B 2013F550       (for tm11/ts11/tu80 tapes on 780 uba0)
>>> D/G B FFF550         (for tm11/ts11/tu80 tapes on 750 or 730 uba0)
```

Then start the bootstrap program with

```
>>> S 0
```

The console should type

```
=
```

You are now talking to the tape bootstrap monitor. At any point in the following procedure you can return to this section, reload the tape bootstrap, and restart the procedure. The console monitor is identical to that loaded from a TU58 console cassette, follow the instructions in section 2 as they apply to this device. The only exception is that when using programs loaded from the tape bootstrap monitor, programs will always return to the monitor (the "=" prompt). This saves your having to type in the above toggle-in code for each program to be loaded.

APPENDIX C — INSTALLATION TROUBLESHOOTING

This appendix lists and explains certain problems which might be encountered while trying to install the 4.2BSD distribution. The information provided here is limited to the early steps in the installation process; i.e. up to the point where the root file system is installed. If you have a problem installing the release consult this section for an indication of the problem before contacting our group.

Using the distribution console medium.

This section describes problems which may occur when using the programs provided on the distributed console medium: TU58 cassette or RX01 floppy disk.

program can not be loaded.

Check to make sure the correct floppy or cassette is being used. If using a floppy, be sure it is not in upside down. If using a cassette on an 11/730, be certain drive 0 is being used. If a hard i/o error occurred while reading a floppy, try reseting the console LSI-11 by powering it on and off. If you can not boot the cassette's bootstrap monitor, verify the standard DEC console cassette can be read; if it can not, your cassette is broken — not uncommon.

program halts without warning.

Check to make sure you have specified the correct disk to format; consult sections 1.3 and 1.4 for a discussion of the VAX and UNIX device naming conventions. On 11/750's, specifying a non-existent MASSBUS device will cause the program to halt as it receives an interrupt (standalone programs operate by polling devices).

If using a floppy, try reading the floppy under your current system. If this works, copy the floppy to a new one and begin again. If using a cassette on an 11/730, do likewise.

format prints "Known devices are ...".

You have requested *format* to work on a device for which it has no driver, or which does not exist; only the indicated devices are supported.

format, boot, or copy prints "unknown drive type".

A MASSBUS disk was specified, but the associated MASSBUS drive type register indicates a drive of unknown type. This probably means you typed something wrong or your hardware is incorrectly configured.

format, boot, or copy prints "unknown device".

The device specified is probably not one of those supported by the distribution; consult section 1.1. If the device is listed in section 1.1, the drive may be dual-ported, or for some other reason the driver was unable to decipher its characteristics. If this is a MASSBUS drive, try powering the MASSBUS adapter and/or controller on and off to clear the drive type register.

copy does not copy 205 records

If a tape read error occurred, clean your tape drive heads. If a disk write error occurred, the disk formatting may have failed. If the disk pack is removable, try another one. If you are currently running UNIX, you can reboot your old system and use *dd* to copy the mini-root file system into a disk partition (assuming the destination is not in use by the running system).

boot prints "not a directory"

The *boot* program was unable to find the requested program because it encountered something other than a directory while searching the file system. This usually indicates no file system is present on the disk partition supplied, or the file system has been corrupted. First check to make sure you typed the correct line to boot. If this is the case and you are booting off the mini-root file system, the mini-root was probably not copied correctly off the tape (perhaps it was not placed in the correct disk partition). Try reinstalling the mini-root file system or, if trying to boot the true root file system, try booting off the mini-root file system and run *fsck* on

September 22, 1983

the restored root file system to insure its integrity. Finally, as a last resort, copy the *boot* program from the mini-root file system to the newly installed root file system.

boot prints "bad format"

The program you requested *boot* to load did not have a 407, 410, or 413 magic number in its header. This should never happen on a distribution system. If you were trying to boot off the root file system, reboot the system on the mini-root file system and look at the program on the root file system. Try copying the copy of vmunix on the mini-root to the root file system also.

boot prints "read short"

The file header for the program indicated a size larger than the actual size of the file located on disk. This is probably the result of file system corruption (or a disk i/o error). Try booting again or creating a new copy of the program to be loaded (see above).

Booting the generic system

This section contains common problems encountered when booting the generic version of the system.

system panics with "panic: iinit"

This occured because the system was unable to locate the program /etc/init. The root file system supplied at the "root device?" prompt was probably incorrect. Remember that when running on the mini-root file system, this question must be answered with something of the form "hp0*". If the answer had been "hp0", the system would have used the "a" partition on unit 0 of the "hp" drive, where presumably no file system exists.

Alternatively, the file system on which you were trying to run is corrupted, or simply missing /etc/init. Try reinstalling the appropriate file system or installing a version of init.

system selects incorrect root device

That is, you try to boot the system single user with "B/2" or "B xxS" but do not get the root file system in the expected location. This is most likely caused by your having many disks available more suited to be a root file system than the one you wanted. For example, if you have a "up" disk and an "hk" disk and install the system on the "hk", then try and boot the system to single-user mode, the heuristic used by the generic system to select the root file system will choose the "up" disk. The following list gives, in descending order, those disks thought most suitable to be a root file system: "hp", "up", "ra", "rb", "rl", "hk" (the position of "rl" is subject to argument). To get the root device you want you must boot using "B/3" or "B ANY", then supply the root device at the prompt.

system crashes during autoconfiguration

This is almost always caused by an unsupported UNIBUS device being present at a location where a supported device was expected. You must disable the device in some way, either by pulling it off the bus, or by moving the location of the console status register (consult Appendix A for a complete list of UNIBUS csr's used in the generic system).

system does not find device(s)

The UNIBUS device is not at a standard location. Consult the list of control status register addresses in Appendix A, or wait to configure a system to your hardware.

Alternatively, certain devices are difficult to locate during autoconfiguration. A classic example is the TS11 tape drive which does not autoconfigure properly if it is rewinding when the system is rebooted. Tape drives should configure properly if they are off-line, or are not performing a tape movement. Disks which are dual-ported should autoconfigure properly if the drive is not being simultaneously accessed through the alternate port.

September 22, 1983

Building console cassettes

This sections describes common problems encountered while constructing a console bootstrap cassette.

system crashes

You are trying to build a cassette for an 11/750. On an 11/750 the system is booted by using a bootstrap prom and sector 0 of the root file system. Refer to section 2.1.5 or *tu* (4) for the appropriate reprimand.

system hangs

You are using an MRSP prom on an 11/750 and think you can ignore the instructions in this document. The problem here is that the generic system only supports the MRSP prom on an 11/730. Using it on an 11/750 requires a special system configuration; consult *tu* (4) for more information.

Building 4.2BSD UNIX† Systems with Config
June, 1983

Samuel J. Leffler

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720
(415) 642-7780

ABSTRACT

This document describes the use of *config*(8) to configure and create bootable 4.2BSD system images. It discusses the structure of system configuration files and how to configure systems with non-standard hardware configurations. Sections describing the preferred way to add new code to the system and how the system's autoconfiguration process operates are included. An appendix contains a summary of the rules used by the system in calculating the size of system data structures, and also indicates some of the standard system size limitations (and how to change them).

---

†UNIX is a Trademark of Bell Laboratories.

July 27, 1983

1. INTRODUCTION

Config is a tool used in building 4.2BSD system images. It takes a file describing a system's tunable parameters and hardware support, and generates a collection of files which are then used to build a copy of UNIX appropriate to that configuration. *Config* simplifies system maintenance by isolating system dependencies in a single, easy to understand, file.

This document describes the content and format of system configuration files and the rules which must be followed when creating these files. Example configuration files are constructed and discussed.

Later sections suggest guidelines to be used in modifying system source and explain some of the inner workings of the autoconfiguration process. Appendix D summarizes the rules used in calculating the most important system data structures and indicates some inherent system data structure size limitations (and how to go about modifying them).

2. CONFIGURATION FILE CONTENTS

A system configuration must include at least the following pieces of information:

- machine type
- cpu type
- system identification
- timezone
- maximum number of users
- location of the root file system
- available hardware

Config allows multiple system images to be generated from a single configuration description. Each system image is configured for identical hardware, but may have different locations for the root file system and, possibly, other system devices.

2.1. Machine type

The *machine type* indicates if the system is going to operate on a DEC VAX-11 computer, or some other machine on which 4.2BSD operates. The machine type is used to locate certain data files which are machine specific and, also, to select rules used in constructing the resultant configuration files.

2.2. Cpu type

The *cpu type* indicates which, of possibly many, cpu's the system is to operate on. For example, if the system is being configured for a VAX-11, it could be running on a VAX-11/780, VAX-11/750, or VAX-11/730. Specifying more than one cpu type implies the system should be configured to run on all the cpu's specified. For some types of machines this is not possible and *config* will print a diagnostic indicating such.

2.3. System identification

The *system identification* is a moniker attached to the system, and often the machine on which the system is to run. For example, at Berkeley we have machines named Ernie (Co-VAX), Kim (No-VAX), and so on. The system identifier selected is used to create a global C "#define" which may be used to isolate system dependent pieces of code in the kernel. For example, Ernie's Varian driver used to be special cased because its interrupt vectors were wired together. The code in the driver which understood how to handle this non-standard hardware configuration was conditionally compiled in only if the system was for Ernie.

The system identifier "GENERIC" is given to a system which will run on any cpu of a particular machine type; it should not otherwise be used for a system identifier.

2.4. Timezone

The timezone in which the system is to run is used to define the information returned by the *gettimeofday* (2) system call. This value is specified as the number of hours east or west of GMT. Negative numbers indicate a value east of GMT. The timezone specification may also indicate the type of daylight savings time rules to be applied.

2.5. Maximum number of users

The system allocates many system data structures at boot time based on the maximum number of users the system will support. This number is normally between 8 and 40, depending on the hardware and expected job mix. The rules used to calculate system data structures are discussed in Appendix D.

2.6. Root file system location

When the system boots it must know the location of the root of the file system tree. This location and the part(s) of the disk(s) to be used for paging and swapping must be specified in order to create a complete configuration description. *Config* uses many rules to calculate default locations for these items; these are described in Appendix B.

When a generic system is configured, the root file system is left undefined until the system is booted. In this case, the root file system need not be specified, only that the system is a generic system.

2.7. Hardware devices

When the system boots it goes through an *autoconfiguration* phase. During this period, the system searches for all those hardware devices which the system builder has indicated might be present. This probing sequence requires certain pieces of information such as register addresses, bus interconnects, etc. A system's hardware may be configured in a very flexible manner or be specified without any flexibility whatsoever. Most people do not configure hardware devices into the system unless they are currently present on the machine, expect them to be present in the near future, or are simply guarding against a hardware failure somewhere else at the site (it is often wise to configure in extra disks in case an emergency requires moving one off a machine which has hardware problems).

The specification of hardware devices usually occupies the majority of the configuration file. As such, a large portion of this document will be spent understanding it. Section 6.3 contains a description of the autoconfiguration process, as it applies to those planning to write, or modify existing, device drivers.

2.8. Optional items

Other than the mandatory pieces of information described above, it is also possible to include various optional system facilities. For example, 4.2BSD can be configured to support binary compatibility for programs built under 4.1BSD. Also, optional support is provided for disk quotas and tracing the performance of the virtual memory subsystem. Any optional facilities to be configured into the system are specified in the configuration file. The resultant files generated by *config* will automatically include the necessary pieces of the system.

July 27, 1983

3. SYSTEM BUILDING PROCESS

In this section we consider the steps necessary to build a bootable system image. We assume the system source is located in the "/sys" directory and that, initially, the system is being configured from source code.

Under normal circumstances there are 5 steps in building a system.

1) Create a configuration file for the system.

2) Make a directory for the system to be constructed in.

3) Run *config* on the configuration file to generate the files required to compile and load the system image.

4) Construct the source code interdependency rules for the configured system.

5) Compile and load the system with *make* (1).

Steps 1 and 2 are usually done only once. When a system configuration changes it usually suffices to just run *config* on the modified configuration file, rebuild the source code dependencies, and remake the system. Sometimes, however, configuration dependencies may not be noticed in which case it is necessary to clean out the relocatable object files saved in the system's directory; this will be discussed later.

3.1. Creating a configuration file

Configuration files normally reside in the directory "/sys/conf". A configuration file is most easily constructed by copying an existing configuration file and modifying it. The 4.2BSD distribution contains a number of configuration files for machines at Berkeley, one may be suitable or, in worst case, you may take the generic configuration file and edit that.

The configuration file must have the same name as the directory in which the configured system is to be built. Further, *config* assumes this directory is located in the parent directory of the directory in which it is run. For example, the generic system has a configuration file "/sys/conf/GENERIC" and an accompanying directory named "/sys/GENERIC". In general it is unwise to move your configuration directories out of "/sys" as most of the system code and the files created by *config* use pathnames of the form "../". If you are running out of space on the file system where the configuration directories are located there is a mechanism for sharing relocatable object files between systems; this is described later.

When building your configuration file, be sure to include the items described in section 2. In particular, the machine type, cpu type, timezone, system identifier, maximum users, and root device must be specified. The specification of the hardware present may take a bit of work; particularly if your hardware is configured at non-standard places (e.g. device registers located at funny places or devices not supported by the system). Section 4 of this document gives a detailed description of the configuration file syntax, section 5 explains some sample configuration files, and section 6 discusses how to add new devices to the system. If the devices to be configured are not already described in one of the existing configuration files you should check the manual pages in section 4 of the UNIX Programmers Manual. For each supported device, the manual page synopsis entry gives a sample configuration line.

Once the configuration file is complete, run it through *config* and look for any errors. Never try and use a system which *config* has complained about; the results are unpredictable. For the most part, *config*'s error diagnostics are self explanatory. It may be the case that the line numbers given with the error messages are off by one.

A successful run of *config* on your configuration file will generate a number of files in the configuration directory. These files are:

• A file to be used by *make* (1) in compiling and loading the system.

July 27, 1983

- One file for each possible system image for your machine which describes where swapping, the root file system, and other miscellaneous system devices are located.

- A collection of header files, one per possible device the system supports, which define the hardware configured.

- A file containing the i/o configuration tables used by the system during its *autoconfiguration* phase.

- An assembly language file of interrupt vectors which connect interrupts from your machine's external buses to the main system path for handling interrupts.

Unless you have reason to doubt *config*, or are curious how the system's autoconfiguration scheme works, you should never have to look at any of these files.

3.2. Constructing source code dependencies

When *config* is done generating the files needed to compile and link your system it will terminate with a message of the form "Don't forget to run make depend". This is a reminder that you should change over to the configuration directory for the system just configured and type "make depend" to build the rules used by *make* to recognize interdependencies in the system source code. This will insure that any changes to a piece of the system source code will result in the proper modules being recompiled the next time *make* is run.

This step is particularly important if your site makes changes to the system include files. The rules generated specify which source code files are dependent on which include files. Without these rules, *make* will not recognize when it must rebuild modules due to a system header file being modified. Note that dependency rules created by this step only reflect directly included files. That is, if file "a" includes another file "b", which includes yet another, say "c", and then "c" is modified, *make* will not recognize that "a" should be recompiled. It is best to keep include file dependencies only one level deep.

3.3. Building the system

The makefile constructed by *config* should allow a new system to be rebuilt by simply typing "make image-name". For example, if you have named your bootable system image "vmunix", then "make vmunix" will generate a bootable image named "vmunix". Alternate system image names are used when the root file system location and/or swapping configuration is done in more than one way. The makefile which *config* creates has entry points for each system image defined in the configuration file. Thus, if you have configured "vmunix" to be a system with the root file system on an "hp" device and "hkvmunix" to be a system with the root file system on an "hk" device, then "make vmunix hkvmunix" will generate binary images for each.

Note that the name of a bootable image is different from the system identifier. All bootable images are configured for the same system; only the information about the root file system and paging devices differ. (This is described in more detail in section 4.)

The last step in the system building process is to rearrange certain commonly used symbols in the symbol table of the system image; the makefile generated by *config* does this automatically for you. This is advantageous for programs such as *ps*(1) and *vmstat*(1), which run much faster when the symbols they need are located at the front of the symbol table. Remember also that many programs expect the currently executing system to be named "/vmunix". If you install a new system and name it something other than "/vmunix", many programs are likely to give strange results.

3.4. Sharing object modules

If you have many systems which are all built on a single machine there are at least two approaches to saving time in building system images. The best way is to have a single system image which is run on all machines. This is attractive since it minimizes disk space used and time required to rebuild systems after making changes. However, it is often the case that one

or more systems will require a separately configured system image. This may be due to limited memory (building a system with many unused device drivers can be expensive), or to configuration requirements (one machine may be a development machine where disk quotas are not needed, while another is a production machine where they are), etc. In these cases it is possible for common systems to share relocatable object modules which are not configuration dependent; most of the module in the directory "/sys/sys" are of this sort.

To share object modules, a generic system should be built. Then, for each system configure the system as before, but before recompiling and linking the system, type "make links". This will cause the system to be searched for source modules which are safe to share between systems and generate symbolic links in the current directory to the appropriate object modules in the directory "../GENERIC". A shell script, "makelinks" is generated with this request and may be checked for correctness. The file "/sys/conf/defines" contains a list of symbols which we believe are safe to ignore when checking the source code for modules which may be shared. Note that this list includes the definitions used to conditionally compile in the virtual memory tracing facilities, and the trace point support used only rarely (even at Berkeley). It may be necessary to modify this file to reflect local needs. Note further, that as described previously, interdependencies which are not directly visible in the source code are not caught. This means that if you place per-system dependencies in an include file, they will not be recognized and the shared code may be selected in an unexpected fashion.

3.5. Building profiled systems

It is simple to configure a system which will automatically collect profiling information as it operates. The profiling data may be collected with *kgmon*(8) and processed with *gprof*(1) to obtain information regarding the system's operation. Profiled systems maintain histograms of the program counter as well as the number of invocations of each routine. The *gprof*(1) command will also generate a dynamic call graph of the executing system and propagate time spent in each routine along the arcs of the call graph (consult the gprof documentation for elaboration). The program counter sampling can be driven by the system clock, or if you have an alternate real time clock this can be used. The latter is highly recommended as use of the system clock will result in statistical anomalies and time spent in the clock routine will not be accurately accounted for.

To configure a profiled system, the −p option should be supplied to *config*. A profiled system is about 5-10% larger in its text space due to the calls to count the subroutine invocations. When the system executes, the profiling data is stored in a buffer which is 1.2 times the size of the text space. The overhead for running a profiled system varies; under normal load we see anywhere from 5-25% of the system time spent in the profiling code.

Note that systems configured for profiling should not be shared as described above unless all the other shared systems are also to be profiled.

4. CONFIGURATION FILE SYNTAX

In this section we consider the specific rules used in writing a configuration file. A complete grammar for the input language can be found in Appendix A and may be of use if you should have problems with syntax errors.

A configuration file is broken up into three logical pieces:

- configuration parameters global to all system images specified in the configuration file,
- parameters specific to each system image to be generated, and
- device specifications.

4.1. Global configuration parameters

The global configuration parameters are the type of machine, cpu types, options, timezone, system identifier, and maximum users. Each is specified with a separate line in the configuration file.

machine *type*

> The system is to run on the machine type specified. No more than one machine type can appear in the configuration file. Legal values are **vax** and **sun**.

cpu "*type*"

> This system is to run on the cpu type specified. More than one cpu type specification can appear in a configuration file. Legal types for a **vax** machine are **VAX780**, **VAX750**, and **VAX730**.

options *optionlist*

> Compile the listed optional code into the system. Options in this list are separated by commas. Possible options are listed at the top of the generic makefile. A line of the form "options FUNNY,HAHA" generates global "#define"s −DFUNNY −DHAHA in the resultant makefile. An option may be given a value by following its name with "=", then the value enclosed in (double) quotes. None of the standard options use such a value. The following options are currently in use: COMPAT (include code for compatiblity with 4.1BSD binaries), INET (Internet communication protocols), PUP (support for a PUP raw interface), and QUOTA (enable disk quotas). There are additional options which are associated with certain peripheral devices; those are listed in the Synopsis section of the manual page for the device.

timezone *number* [**dst** [*number*]]

> Specifies the timezone you are in. This is measured in the number of hours your timezone is west of GMT. EST is 5 hours west of GMT, PST is 8. Negative numbers indicate hours east of GMT. If you specify **dst**, the system will operate under daylight savings time. An optional integer or floating point number may be included to specify a particular daylight saving time correction algorithm; the default value is 1, indicating the United States. Other values are: 2 (Australian style), 3 (Western European), 4 (Middle European), and 5 (Eastern European). See *gettimeofday* (2) and *ctime* (3) for more information.

ident *name*

> This system is to be known as *name*. This is usually a cute name like ERNIE (short for Ernie Co-Vax) or VAXWELL (for Vaxwell Smart).

maxusers *number*

> The maximum expected number of simultaneously active user on this system is *number*. This number is used to size several system data structures.

4.2. System Image parameters

Multiple bootable images may be specified in a single configuration file. The systems will have the same global configuration parameters and devices, but the location of the root file system and other system specific devices may be different. A system image is specified with a "config" line:

config *sysname config-clauses*

The *sysname* field is the name given to the loaded system image; almost everyone names their standard system image "vmunix". The configuration clauses are one or more specifications indicating where the root file system is located, how many paging devices there are and where they go. The device used by the system to process argument lists during *execve*(2) calls may also be specified, though in practice this is almost always selected by *config* using one of its rules for selecting default locations for system devices.

A configuration clause is one of the following

root [**on**] *root-device*
swap [**on**] *swap-device* [**and** *swap-device*]
dumps [**on**] *dump-device*
args [**on**] *arg-device*

(the "on" is optional.) Multiple configuration clauses are separated by white space; *config* allows specifications to be continued across multiple lines by beginning the continuation line with a tab character. The "root" clause specifies where the root file system is located, the "swap" clause indicates swapping and paging area(s), the "dumps" clause can be used to force system dumps to be taken on a particular device, and the "args" clause can be used to specify that argument list processing for *execve* should be done on a particular disk.

The device names supplied in the clauses may be fully specified as a device, unit, and file system partition; or underspecified in which case *config* will use builtin rules to select default unit numbers and file system partitions. The defaulting rules are a bit complicated as they are dependent on the overall system configuration. For example, the swap area need not be specified at all if the root device is specified; in this case the swap area is placed in the "b" partition of the same disk where the root file system is located. Appendix B contains a complete list of the defaulting rules used in selecting system configuration devices.

The device names are translated to the appropriate major and minor device numbers on a per-machine basis. A file, "/sys/conf/devices.machine" (where "machine" is the machine type specified in the configuration file), is used to map a device name to its major block device number. The minor device number is calculated using the standard disk partitioning rules: on unit 0, partition "a" is minor device 0, partition "b" is minor device 1, and so on; for units other than 0, add 8 times the unit number to get the minor device.

If the default mapping of device name to major/minor device number is incorrect for your configuration, it can be replaced by an explicit specification of the major/minor device. This is done by substituting

major *x* **minor** *y*

where the device name would normally be found. For example,

config vmunix root on major 99 minor 1

Normally, the areas configured for swap space are sized by the system at boot time. If a non-standard partition size is to be used for one or more swap areas, this can also be specified. To do this, the device name specified for a swap area should have a "size" specification appended. For example,

config vmunix root on hp0 swap on hp0b size 1200

would force swapping to be done in partition "b" of "hp0" and the swap partition size would be set to 1200 sectors. A swap area sized larger than the associated disk partition is trimmed to the partition size.

July 27, 1983

To create a generic configuration, only the clause "swap generic" should be specified; any extra clauses will cause an error.

4.3. Device specifications

Each device attached to a machine must be specified to *config* so that the system generated will know to probe for it during the autoconfiguration process carried out at boot time. Hardware specified in the configuration need not actually be present on the machine where the generated system is to be run. Only the hardware actually found at boot time will be used by the system.

The specification of hardware devices in the configuration file parallels the interconnection hierarchy of the machine to be configured. On the VAX, this means a configuration file must indicate what MASSBUS and UNIBUS adapters are present, and to which *nexi* they might be connected*. Similarly, devices and controllers must be indicated as possibly being connected to one or more adapters. A device description may provide a complete definition of the possible configuration parameters or it may leave certain parameters undefined and make the system probe for all the possible values. The latter allows a single device configuration list to match many possible physical configurations. For example, a disk may be indicated as present at UNIBUS adapter 0, or at any UNIBUS adapter which the system locates at boot time. The latter scheme, termed *wildcarding*, allows more flexibility in the physical configuration of a system; if a disk must be moved around for some reason, the system will still locate it at the alternate location.

A device specification takes one of the following forms:

master *device-name device-info*
controller *device-name device-info* [*interrupt-spec*]
device *device-name device-info interrupt-spec*
disk *device-name device-info*
tape *device-name device-info*

A "master" is a MASSBUS tape controller; a "controller" is a disk controller, a UNIBUS tape controller, a MASSBUS adapter, or a UNIBUS adapter. A "device" is an autonomous device which connects directly to a UNIBUS adapter (as opposed to something like a disk which connects through a disk controller). "Disk" and "tape" identify disk drives and tape drives connected to a "controller" or "master".

The *device-name* is one of the standard device names, as indicated in section 4 of the UNIX Programmers Manual, concatenated with the *logical* unit number to be assigned the device (the *logical* unit number may be different than the *physical* unit number indicated on the front of something like a disk; the *logical* unit number is used to refer to the UNIX device, not the physical unit number). For example, "hp0" is logical unit 0 of a MASSBUS storage device, even though it might be physical unit 3 on MASSBUS adapter 1.

The *device-info* clause specifies how the hardware is connected in the interconnection hierarchy. On the VAX, UNIBUS and MASSBUS adapters are connected to the internal system bus through a *nexus*. Thus, one of the following specifications would be used:

controller	mba0	**at nexus** *x*
controller	uba0	**at nexus** *x*

To tie a controller to a specific nexus, "x" would be supplied as the number of that nexus; otherwise "x" may be specified as "?", in which case the system will probe all nexi present looking for the specified controller.

The remaining interconnections on the VAX are:

---

* While VAX-11/750's and VAX-11/730 do not actually have nexi, the system treats them as having *simulated nexi* to simplify device configuration.

- a controller may be connected to another controller (e.g. a disk controller attached to a UNIBUS adapter),
- a master is always attached to a controller (a MASSBUS adaptor),
- a tape is always attached to a master (for MASSBUS tape drives),
- a disk is always attached to a controller, and
- devices are always attached to controllers (e.g. UNIBUS controllers attached to UNIBUS adapters).

The following lines give an example of each of these interconnections:

controller	hk0	at uba0 ...
master	ht0	at mba0 ...
tape	tu0	at ht0 ...
disk	rk1	at hk0 ...
device	dz0	at uba0 ...

Any piece of hardware which may be connected to a specific controller may also be wildcarded across multiple controllers.

The final piece of information needed by the system to configure devices is some indication of where or how a device will interrupt. For tapes and disks, simply specifying the *slave* or *drive* number is sufficient to locate the control status register for the device. For controllers, the control status register must be given explicitly, as well how many interrupt vectors are used and the names of the routines to which they should be bound. Thus the example lines given above might be completed as:

controller	hk0	at uba0 csr 0177440	vector rkintr
master	ht0	at mba0 drive 0	
tape	tu0	at ht0 slave 0	
disk	rk1	at hk0 drive 1	
device	dz0	at uba0 csr 0160100	vector dzrint dzxint

Certain device drivers require extra information passed to them at boot time to tailor their operation to the actual hardware present. The line printer driver, for example, needs to know how many columns are present on each non-standard line printer (i.e. a line printer with other than 80 columns). The drivers for the terminal multiplexors need to know which lines are attached to modem lines so that no one will be allowed to use them unless a connection is present. For this reason, one last parameter may be specified to a *device*, a *flags* field. It has the syntax

flags *number*

and is usually placed after the *csr* specification. The *number* is passed directly to the associated driver. The manual pages in section 4 should be consulted to determine how each driver uses this value (if at all). Communications interface drivers commonly use the flags to indicate whether modem control signals are in use.

The exact syntax for each specific device is given in the Synopsis section of its manual page in section 4 of the manual.

4.4. Pseudo-devices

A number of drivers and software subsystems are treated like device drivers without any associated hardware. To include any of these pieces, a "pseudo-device" specification must be used. A specification for a pseudo device takes the form

pseudo-device *device-name* [*howmany*]

Examples of pseudo devices are bk, the Berknet line discipline, pty, the pseudo terminal driver (where the optional *howmany* value indicates the number of pseudo terminals to configure, 32 default), and inet, the DARPA Internet protocols (one must also specify INET in the "options"). Other pseudo devices for the network include loop, the software loopback

interface, **imp** (required when a CSS or ACC imp is configured), and **ether** (used by the Address Resolution Protocol on 10 Mb/sec ethernets). More information on configuring each of these can also be found in section 4 of the manual.

5. SAMPLE CONFIGURATION FILES

In this section we will consider how to configure a sample VAX-11/780 system on which the hardware can be reconfigured to guard against various hardware mishaps. We then study the rules needed to configure a VAX-11/750 to run in a networking environment.

5.1. VAX-11/780 System

Our VAX-11/780 is configured with hardware recommended in the document "Hints on Configuring a VAX for 4.2BSD" (this is one of the high-end configurations). Table 1 lists the pertinent hardware to be configured.

Item	Vendor	Connection	Name	Reference
cpu	DEC		VAX780	
MASSBUS controller	Emulex	nexus ?	mba0	hp(4)
disk	Fujitsu	mba0	hp0	
disk	Fujitsu	mba0	hp1	
MASSBUS controller	Emulex	nexus ?	mba1	
disk	Fujitsu	mba1	hp2	
disk	Fujitsu	mba1	hp3	
UNIBUS adapter	DEC	nexus ?		
tape controller	Emulex	uba0	tm0	tm(4)
tape drive	Kennedy	tm0	te0	
tape drive	Kennedy	tm0	te1	
terminal multiplexor	Emulex	uba0	dh0	dh(4)
terminal multiplexor	Emulex	uba0	dh1	
terminal multiplexor	Emulex	uba0	dh2	

Table 1. VAX-11/780 Hardware support.

We will call this machine ANSEL and construct a configuration file one step at a time.

The first step is to fill in the global configuration parameters. The machine is a VAX, so the *machine type* is "vax". We will assume this system will run only on this one processor, so the *cpu type* is "VAX780". The options are empty since this is going to be a "vanilla" VAX. The system identifier, as mentioned before, is "ANSEL" and the maximum number of users we plan to support is about 40. Thus the beginning of the configuration file looks like this:

```
#
# ANSEL VAX (a picture perfect machine)
#
machine            vax
cpu                VAX780
timezone           8 dst
ident              ANSEL
maxusers           40
```

To this we must then add the specifications for three system images. The first will be our standard system with the root on "hp0" and swapping on the same drive as the root. The second will have the root file system in the same location, but swap space interleaved among drives on each controller. Finally, the third will be a generic system, to allow us to boot off any of the four disk drives.

config	vmunix	root on hp0
config	hpvmunix	root on hp0 swap on hp0 and hp2
config	genvmunix	swap generic

Finally, the hardware must be specified. Let us first just try transcribing the information from Table 1.

controller	mba0	at nexus ?	
disk	hp0	at mba0 disk 0	
disk	hp1	at mba0 disk 1	
controller	mba1	at nexus ?	
disk	hp2	at mba1 disk 2	
disk	hp3	at mba1 disk 3	
controller	uba0	at nexus ?	
controller	tm0	at uba0 csr 0172520	vector tmintr
tape	te0	at tm0 drive 0	
tape	te1	at tm0 drive 1	
device	dh0	at uba0 csr 0160020	vector dhrint dhxint
device	dm0	at uba0 csr 0170500	vector dmintr
device	dh1	at uba0 csr 0160040	vector dhrint dhxint
device	dh2	at uba0 csr 0160060	vector dhrint dhxint

(Oh, I forgot to mention one panel of the terminal multiplexor has modem control, thus the "dm0" device.)

This will suffice, but leaves us with little flexibility. Suppose our first disk controller were to break. We would like to recable the drives normally on the second controller so that all our disks could still be used without reconfiguring the system. To do this we wildcard the MASSBUS adapter connections and also the slave numbers. Further, we wildcard the UNIBUS adapter connections in case we decide some time in the future to purchase another adapter to offload the single UNIBUS we currently have. The revised device specifications would then be:

controller	mba0	at nexus ?	
disk	hp0	at mba? disk ?	
disk	hp1	at mba? disk ?	
controller	mba1	at nexus ?	
disk	hp2	at mba? disk ?	
disk	hp3	at mba? disk ?	
controller	uba0	at nexus ?	
controller	tm0	at uba? csr 0172520	vector tmintr
tape	te0	at tm0 drive 0	
tape	te1	at tm0 drive 1	
device	dh0	at uba? csr 0160020	vector dhrint dhxint
device	dm0	at uba? csr 0170500	vector dmintr
device	dh1	at uba? csr 0160040	vector dhrint dhxint
device	dh2	at uba? csr 0160060	vector dhrint dhxint

The completed configuration file for ANSEL is shown in Appendix C.

5.2. VAX-11/750 with network support

Our VAX-11/750 system will be located on two 10Mb/s Ethernet local area networks and also the DARPA Internet. The system will have a MASSBUS drive for the root file system and two UNIBUS drives. Paging is interleaved among all three drives. We have sold our standard DEC terminal multiplexors since this machine will be accessed solely through the network. This machine is not intended to have a large user community, it does not have a great deal of memory. First the global parameters:

July 27, 1983

```
#
# UCBVAX (Gateway to the world)
#
machine            vax
cpu                "VAX780"
cpu                "VAX750"
ident              UCBVAX
timezone           8 dst
maxusers           32
options            INET
```

The multiple cpu types allow us to replace UCBVAX with a more powerful cpu without reconfiguring the system. The value of 32 given for the maximum number of users is done to force the system data structures to be over-allocated. That is desirable on this machine because, while it is not expected to support many users, it is expected to perform a great deal of work. Upping this value results in a larger disk buffer cache than would normally be allocated if the true number of users were given. The "INET" indicates we plan to use the DARPA standard Internet protocols on this machine.

The system images and disks are configured in next.

```
config         vmunix     root on hp swap on hp and rk0 and rk1
config         upvmunix   root on up
config         hkvmunix   root on hk swap on rk0 and rk1

controller     mba0       at nexus ?
controller     uba0       at nexus ?
disk           hp0        at mba? drive 0
disk           hp1        at mba? drive 1
controller     sc0        at uba? csr 0176700     vector upintr
disk           up0        at sc0 drive 0
disk           up1        at sc0 drive 1
controller     hk0        at uba? csr 0177440     vector rkintr
disk           rk0        at hk0 drive 0
disk           rk1        at hk0 drive 1
```

UCBVAX requires heavy interleaving of its paging area to keep up with all the mail traffic it handles. The limiting factor on this system's performance is usually the number of disk arms, as opposed to memory or cpu cycles. The extra UNIBUS controller, "sc0", is in case the MASSBUS controller breaks and a spare controller must be *installed* (most of our old UNIBUS controllers have been *replaced* with the newer MASSBUS controllers, so we have a number of these around as spares).

Finally, we add in the network support. The Internet protocols require an "inet" *pseudo-device* in addition to the global "INET" option specified above. Pseudo terminals are needed to allow users to log in across the network (remember the only hardwired terminal is the console). The connection to the Internet is through an IMP, this requires yet another *pseudo-device* (in addition to the actual hardware device used by the IMP software). And, finally, there are the two Ethernet devices. These use a special protocol, the Address Resolution Protocol (ARP), to map between Internet and Ethernet addresses. Thus, yet another *pseudo-device* is needed. The additional device specifications are show below.

6. ADDING NEW SYSTEM SOFTWARE

This section is not for the novice, it describes some of the inner workings of the configuration process as well as the pertinent parts of the system autoconfiguration process. It is intended to give those people who intend to install new device drivers and/or other system facilities sufficient information to do so in the manner which will allow others to easily share the changes.

This section is broken into four parts:

- general guidelines to be followed in modifying system code,
- how to add a device driver to 4.2BSD,
- how UNIBUS device drivers are autoconfigured under 4.2BSD on the VAX, and
- how to add non-standard system facilities to 4.2BSD.

6.1. Modifying system code

If you wish to make site-specific modifications to the system it is best to bracket them with

```
#ifdef SITENAME
...
#endif
```

to allow your source to be easily distributed to others, and also to simplify *diff*(1) listings. If you choose not to use a source code control system (e.g. SCCS, RCS), and perhaps even if you do, it is recommended that you save the old code with something of the form:

```
#ifndef SITENAME
...
#endif
```

We try to isolate our site-dependent code in individual files which may be configured with pseudo-device specifications.

Indicate machine specific code with "#ifdef vax". 4.2BSD has undergone extensive work to make it extremely portable to machines with similar architectures — you may someday find yourself trying to use a single copy of the source code on multiple machines.

Use *lint* periodically if you make changes to the system. The 4.2BSD release has only one line of lint in it. It is very simple to lint the kernel. Use the LINT configuration file, designed to pull in as much of the kernel source code as possible, in the following manner.

```
$ cd /sys/conf
$ mkdir ../LINT
$ config LINT
$ cd ../LINT
$ make depend
$ make assym.s
$ make −k lint > linterrs 2>&1 &
(or for users of csh (1))
% make −k >& linterrs
```

This takes about 45 minutes on a lightly loaded VAX-11/750, but is well worth it.

6.2. Adding device drivers to 4.2BSD

The i/o system and *config* have been designed to easily allow new device support to be added. As described in "Installing and Operating 4.2BSD on the VAX", the system source directories are organized as follows:

July 27, 1983

/sys/h	machine independent include files
/sys/sys	machine independent system source files
/sys/conf	site configuration files and basic templates
/sys/net	network independent, but network related code
/sys/netinet	DARPA Internet code
/sys/netimp	IMP support code
/sys/netpup	PUP-1 support code
/sys/vax	VAX specific mainline code
/sys/vaxif	VAX network interface code
/sys/vaxmba	VAX MASSBUS device drivers and related code
/sys/vaxuba	VAX UNIBUS device drivers and related code

Existing block and character device drivers for the VAX reside in "/sys/vax", "/sys/vaxmba", and "/sys/vaxuba". Network interface drivers reside in "/sys/vaxif". Any new device drivers should be placed in the appropriate source code directory and named so as not to conflict with existing devices. Normally, definitions for things like device registers are placed in a separate file in the same directory. For example, the "dh" device driver is named "dh.c" and its associated include file is named "dhreg.h".

Once the source for the device driver has been placed in a directory, the file "/sys/conf/files.machine", and possibly "/sys/conf/devices.machine" should be modified. The *files* files in the conf directory contain a line for each source or binary-only file in the system. Those files which are machine independent are located in "/sys/conf/files" while machine specific files are in "/sys/conf/files.machine". The "devices.machine" file is used to map device names to major block device numbers. If the device driver being added provides support for a new disk you will want to modify this file (the format is obvious).

The format of the *files* file has grown somewhat complex over time. Entries are normally of the form

    vaxuba/foo.c   **optional** foo **device-driver**

where the keyword *optional* indicates that to compile the "foo" driver into the system it must be specified in the configuration file. If instead the driver is specified as *standard*, the file will be loaded no matter what configuration is requested. This is not normally done with device drivers. The fact that the file is specified as a *device-driver* results, on the VAX, in the compilation including a $-i$ option for the C optimizer. This is required when pointer references are made to memory locations in the VAX i/o address space.

Aside from including the driver in the *files* file, it must also be added to the device configuration tables. These are located in "/sys/vax/conf.c", or similar for machines other than the VAX. If you don't understand what to add to this file, you should study an entry for an existing driver. Remember that the position in the block device table specifies what the major block device number is, this is needed in the "devices.machine" file if the device is a disk.

With the configuration information in place, your configuration file appropriately modified, and a system reconfigured and rebooted you should incorporate the shell commands needed to install the special files in the file system to the file "/dev/MAKEDEV" or "/dev/MAKEDEV.local". This is discussed in the document "Installing and Operating 4.2BSD on the VAX".

6.3. Autoconfiguration on the VAX

4.2BSD (and 4.1BSD) require all device drivers to conform to a set of rules which allow the system to:

1)   support multiple UNIBUS and MASSBUS adapters,

2)   support system configuration at boot time, and

3)   manage resources so as not to crash when devices request resources which are unavailable.

In addition, devices such as the RK07 which require everyone else to get off the UNIBUS when they are running need cooperation from other DMA devices if they are to work. Since it is unlikely that you will be writing a device driver for a MASSBUS device, this section is devoted exclusively to describing the i/o system and autoconfiguration process as it applies to UNIBUS devices.

Each UNIBUS on a VAX has a set of resources:

•   496 map registers which are used to convert from the 18 bit UNIBUS addresses into the much larger VAX address space.

•   Some number of buffered data paths (3 on an 11/750, 15 on an 11/780, 0 on an 11/730) which are used by high speed devices to transfer data using fewer bus cycles.

There is a structure of type *struct uba_hd* in the system per UNIBUS adapter used to manage these resources. This structure also contains a linked list where devices waiting for resources to complete DMA UNIBUS activity have requests waiting.

There are three central structures in the writing of drivers for UNIBUS controllers; devices which do not do DMA i/o can often use only two of these structures. The structures are *struct uba_ctlr*, the UNIBUS controller structure, *struct uba_device* the UNIBUS device structure, and *struct uba_driver*, the UNIBUS driver structure. The *uba_ctlr* and *uba_device* structures are in one-to-one correspondence with the definitions of controllers and devices in the system configuration. Each driver has a *struct uba_driver* structure specifying an internal interface to the rest of the system.

Thus a specification

        controller sc0 at uba0 csr 0176700 vector upintr

would cause a *struct uba_ctlr* to be declared and initialized in the file *ioconf.c* for the system configured from this description. Similarly specifying

        disk up0 at sc0 drive 0

would declare a related *uba_device* in the same file. The *up.c* driver which implements this driver specifies in its declarations:

        int   upprobe(), upslave(), upattach(), updgo(), upintr();
        struct uba_ctlr *upminfo[NSC];
        struct uba_device *updinfo[NUP];
        u_short upstd[] = { 0776700, 0774400, 0776300, 0 };
        struct uba_driver scdriver =
            { upprobe, upslave, upattach, updgo, upstd, "up", updinfo, "sc", upminfo };

initializing the *uba_driver* structure. The driver will support some number of controllers named *sc0*, *sc1*, etc, and some number of drives named *up0*, *up1*, etc. where the drives may be on any of the controllers (that is there is a single linear name space for devices, separate from the controllers.)

We now explain the fields in the various structures. It may help to look at a copy of *vaxuba/ubareg.h*, *h/ubavar.h* and drivers such as *up.c* and *dz.c* while reading the descriptions of the various structure fields.

uba_driver structure

One of these structures exists per driver. It is initialized in the driver and contains functions used by the configuration program and by the UNIBUS resource routines. The fields of the structure are:

ud_probe

A routine which is given a *caddr_t* address as argument and should cause an interrupt on the device whose control-status register is at that address in virtual memory. It may be the case that the device does not exist, so the probe routine should use delays (via the DELAY(n) macro which delays for *n* microseconds) rather than waiting for specific events to occur. The routine must **not** declare its argument as a *register* parameter, but **must** declare

        register int br, cvec;

as local variables. At boot time the system takes special measures that these variables are "value-result" parameters. The *br* is the IPL of the device when it interrupts, and the *cvec* is the interrupt vector address on the UNIBUS. These registers are actually filled in in the interrupt handler when an interrupt occurs.

As an example, here is the *up.c* probe routine:

```
upprobe(reg)
        caddr_t reg;
{
        register int br, cvec;

#ifdef lint
        br = 0; cvec = br; br = cvec;
#endif
        ((struct updevice *)reg)->upcs1 = UP_IE|UP_RDY;
        DELAY(10);
        ((struct updevice *)reg)->upcs1 = 0;
        return (sizeof (struct updevice));
}
```

The definitions for *lint* serve to indicate to it that the *br* and *cvec* variables are value-result. The statements here interrupt enable the device and write the ready bit UP_RDY. The 10 microsecond delay insures that the interrupt enable will not be canceled before the interrupt can be posted. The return of "sizeof (struct updevice)" here indicates that the probe routine is satisfied that the device is present (the value returned is not currently used, but future plans dictate you should return the amount of space in the device's register bank). A probe routine may use the function "badaddr" to see if certain other addresses are accessible on the UNIBUS (without generating a machine check), or look at the contents of locations where certain registers should be. If the registers contents are not acceptable or the addresses don't respond, the probe routine can return 0 and the device will not be considered to be there.

One other thing to note is that the action of different VAXen when illegal addresses are accessed on the UNIBUS may differ. Some of the machines may generate machine checks and some may cause UNIBUS errors. Such considerations are handled by the configuration program and the driver writer need not be concerned with them.

It is also possible to write a very simple probe routine for a one-of-a-kind device if probing is difficult or impossible. Such a routine would include statements of the form:

        br = 0x15;
        cvec = 0200;

for instance, to declare that the device ran at UNIBUS br5 and interrupted through vector 0200 on the UNIBUS. The current UDA-50 driver does something similar to this because the device is so difficult to force an interrupt on that it hardly seems worthwhile.

ud_slave

This routine is called with a *uba_device* structure (yet to be described) and the address of

the device controller. It should determine whether a particular slave device of a controller is present, returning 1 if it is and 0 if it is not. As an example here is the slave routine for *up.c*.

```
upslave(ui, reg)
      struct uba_device *ui;
      caddr_t reg;
{
      register struct updevice *upaddr = (struct updevice *)reg;

      upaddr->upcs1 = 0;              /* conservative */
      upaddr->upcs2 = ui->ui_slave;
      if (upaddr->upcs2&UPCS2_NED) {
            upaddr->upcs1 = UP_DCLR|UP_GO;
            return (0);
      }
      return (1);
}
```

Here the code fetches the slave (disk unit) number from the *ui_slave* field of the *uba_device* structure, and sees if the controller responds that that is a non-existent driver (NED). If the drive a drive clear is issued to clean the state of the controller, and 0 is returned indicating that the slave is not there. Otherwise a 1 is returned.

ud_attach

The attach routine is called after the autoconfigure code and the driver concur that a peripheral exists attached to a controller. This is the routine where internal driver state about the peripheral can be initialized. Here is the *attach* routine from the *up.c* driver:

```
upattach(ui)
      register struct uba_device *ui;
{
      register struct updevice *upaddr;

      if (upwstart == 0) {
            timeout(upwatch, (caddr_t)0, hz);
            upwstart++;
      }
      if (ui->ui_dk >= 0)
            dk_mspw[ui->ui_dk] = .0000020345;
      upip[ui->ui_ctlr][ui->ui_slave] = ui;
      up_softc[ui->ui_ctlr].sc_ndrive++;
      ui->ui_type = upmaptype(ui);
}
```

The attach routine here performs a number of functions. The first time any drive is attached to the controller it starts the timeout routine which watches the disk drives to make sure that interrupts aren't lost. It also initializes, for devices which have been assigned *iostat* numbers (when ui->ui_dk >= 0), the transfer rate of the device in the array *dk_mspw*, the fraction of a second it takes to transfer 16 bit word. It then initializes an inverting pointer in the array *upip* which will be used later to determine, for a particular *up* controller and slave number, the corresponding *uba_device*. It increments the count of the number of devices on this controller, so that search commands can later be avoided if the count is exactly 1. It then attempts to decipher the actual type of drive attached to the controller in a controller-specific way. On the EMULEX SC-21 it may ask for the number of tracks on the device and use this to decide what the drive type is. The drive type is used to setup disk partition mapping tables and other device specific information.

July 27, 1983

ud_dgo

Is the routine which is called by the UNIBUS resource management routines when an operation is ready to be started (because the required resources have been allocated). The routine in *up.c* is:

```
        updgo(um)
                struct uba_ctlr *um;
        {
                register struct updevice *upaddr = (struct updevice *)um->um_addr;

                upaddr->upba = um->um_ubinfo;
                upaddr->upcs1 = um->um_cmd|((um->um_ubinfo>>8)&0x300);
        }
```

This routine uses the field *um_ubinfo* of the *uba_ctlr* structure which is where the UNIBUS routines store the UNIBUS map allocation information. In particluar, the low 18 bits of this word give the UNIBUS address assigned to the transfer. The assignment to *upba* in the go routine places the low 16 bits of the UNIBUS address in the disk UNIBUS address register. The next assignment places the disk operation command and the extended (high 2) address bits in the device control-status register, starting the i/o operation. The field *um_cmd* was initialized with the command to be stuffed here in the driver code itself before the call to the *ubago* routine which eventually resulted in the call to *updgo*.

ud_addr

Are the conventional addresses for the device control registers in UNIBUS space. This information is used by the system to look for instances of the device supported by the driver. When the system probes for the device it first checks for a control-status register located at the address indicated in the configuration file (if supplied), then uses the list of conventional addresses pointed to be *ud_addr*.

ud_dname

Is the name of a *device* supported by this controller; thus the disks on a SC-21 controller are called *up0*, *up1*, etc. That is because this field contains *up*.

ud_dinfo

Is an array of back pointers to the *uba_device* structures for each device attached to the controller. Each driver defines a set of controllers and a set of devices. The device address space is always one-dimensional, so that the presence of extra controllers may be masked away (e.g. by pattern matching) to take advantage of hardware redundancy. This field is filled in by the configuration program, and used by the driver.

ud_mname

The name of a controller, e.g. *sc* for the *up.c* driver. The first SC-21 is called *sc0*, etc.

ud_minfo

The backpointer array to the structures for the controllers.

ud_xclu

If non-zero specifies that the controller requires exclusive use of the UNIBUS when it is running. This is non-zero currently only for the RK611 controller for the RK07 disks to map around a hardware problem. It could also be used if 6250bpi tape drives are to be used on the UNIBUS to insure that they get the bandwidth that they need (basically the whole bus).

uba_ctlr structure

One of these structures exists per-controller. The fields link the controller to its UNIBUS adapter and contain the state information about the devices on the controller. The fields are:

um_driver

A pointer to the *struct uba_device* for this driver, which has fields as defined above.

um_ctlr

The controller number for this controller, e.g. the 0 in *sc0*.

um_alive

Set to 1 if the controller is considered alive; currently, always set for any structure encountered during normal operation. That is, the driver will have a handle on a *uba_ctlr* structure only if the configuration routines set this field to a 1 and entered it into the driver tables.

um_intr

The interrupt vector routines for this device. These are generated by *config* and this field is initialized in the *ioconf.c* file.

um_hd

A back-pointer to the UNIBUS adapter to which this controller is attached.

um_cmd

A place for the driver to store the command which is to be given to the device before calling the routine *ubago* with the devices *uba_device* structure. This information is then retrieved when the device go routine is called and stuffed in the device control status register to start the i/o operation.

um_ubinfo

Information about the UNIBUS resources allocated to the device. This is normally only used in device driver go routine (as *updgo* above) and occasionally in exceptional condition handling such as ECC correction.

um_tab

This buffer structure is a place where the driver hangs the device structures which are ready to transfer. Each driver allocates a buf structure for each device (e.g. *updtab* in the *up.c* driver) for this purpose. You can think of this structure as a device-control-block, and the buf structures linked to it as the unit-control-blocks. The code for dealing with this structure is stylized; see the *rk.c* or *up.c* driver for the details. If the *ubago* routine is to be used, the structure attached to this *buf* structure must be:

- A chain of *buf* structures for each waiting device on this controller.

- On each waiting *buf* structure another *buf* structure which is the one containing the parameters of the i/o operation.

uba_device structure

One of these structures exist for each device attached to a UNIBUS controller. Devices which are not attached to controllers or which perform no buffered data path DMA i/o may have only a device structure. Thus *dz* and *dh* devices have only *uba_device* structures. The fields are:

ui_driver

A pointer to the *struct uba_driver* structure for this device type.

ui_unit

The unit number of this device, e.g. 0 in **up0**, or 1 in **dh1**.

ui_ctlr

The number of the controller on which this device is attached, or −1 if this device is not on a controller.

ui_ubanum

The number of the UNIBUS on which this device is attached.

ui_slave

The slave number of this device on the controller which it is attached to, or −1 if the device is not a slave. Thus a disk which was unit 2 on a SC-21 would have *ui_slave* 2; it might or might not be *up2*, that depends on the system configuration specification.

July 27, 1983

ui_intr

    The interrupt vector entries for this device, copied into the UNIBUS interrupt vector at boot time. The values of these fields are filled in by *config* to small code segments which it generates in the file *ubglue.s.*

ui_addr

    The control-status register address of this device.

ui_dk

    The iostat number assigned to this device. Numbers are assigned to disks only, and are small positive integers which index the various *dk_** arrays in *<sys/dk.h>*.

ui_flags

    The optional "flags *xxx*" parameter from the configuration specification was copied to this field, to be interpreted by the driver. If *flags* was not specified, then this field will contain a 0.

ui_alive

    The device is really there. Presently set to 1 when a device is determined to be alive, and left 1.

ui_type

    The device type, to be used by the driver internally.

ui_physaddr

    The physical memory address of the device control-status register. This is used in the device dump routines typically.

ui_mi

    A *struct uba_ctlr* pointer to the controller (if any) on which this device resides.

ui_hd

    A *struct uba_hd* pointer to the UNIBUS on which this device resides.

UNIBUS resource management routines

    UNIBUS drivers are supported by a collection of utility routines which manage UNIBUS resources. If a driver attempts to bypass the UNIBUS routines, other drivers may not operate properly. The major routines are: *uballoc* to allocate UNIBUS resources, *ubarelse* to release previously allocated resources, and *ubago* to initiate DMA. When allocating UNIBUS resources you may request that you

NEEDBDP

    if you need a buffered data path,

HAVEBDP

    if you already have a buffered data path and just want new mapping registers (and access to the UNIBUS), and

CANTWAIT

    if you are calling (potentially) from interrupt level

If the presentation here does not answer all the questions you may have, consult the file /sys/vaxuba/uba.c

Autoconfiguration requirements

    Basically all you have to do is write a *ud_probe* and a *ud_attach* routine for the controller. It suffices to have a *ud_probe* routine which just initializes *br* and *cvec*, and a *ud_attach* routine which does nothing. Making the device fully configurable requires, of course, more work, but is worth it if you expect the device to be in common usage and want to share it with others.

    If you managed to create all the needed hooks, then make sure you include the necessary header files; the ones included by *vaxuba/ct.c* are nearly minimal. Order is important here, don't be surprised at undefined structure complaints if you order the includes wrongly. Finally

um_ctlr
  The controller number for this controller, e.g. the 0 in *sc0*.

um_alive
  Set to 1 if the controller is considered alive; currently, always set for any structure encountered during normal operation. That is, the driver will have a handle on a *uba_ctlr* structure only if the configuration routines set this field to a 1 and entered it into the driver tables.

um_intr
  The interrupt vector routines for this device. These are generated by *config* and this field is initialized in the *ioconf.c* file.

um_hd
  A back-pointer to the UNIBUS adapter to which this controller is attached.

um_cmd
  A place for the driver to store the command which is to be given to the device before calling the routine *ubago* with the devices *uba_device* structure. This information is then retrieved when the device go routine is called and stuffed in the device control status register to start the i/o operation.

um_ubinfo
  Information about the UNIBUS resources allocated to the device. This is normally only used in device driver go routine (as *updgo* above) and occasionally in exceptional condition handling such as ECC correction.

um_tab
  This buffer structure is a place where the driver hangs the device structures which are ready to transfer. Each driver allocates a buf structure for each device (e.g. *updtab* in the *up.c* driver) for this purpose. You can think of this structure as a device-control-block, and the buf structures linked to it as the unit-control-blocks. The code for dealing with this structure is stylized; see the *rk.c* or *up.c* driver for the details. If the *ubago* routine is to be used, the structure attached to this *buf* structure must be:

  ● A chain of *buf* structures for each waiting device on this controller.

  ● On each waiting *buf* structure another *buf* structure which is the one containing the parameters of the i/o operation.

uba_device structure

  One of these structures exist for each device attached to a UNIBUS controller. Devices which are not attached to controllers or which perform no buffered data path DMA i/o may have only a device structure. Thus *dz* and *dh* devices have only *uba_device* structures. The fields are:

ui_driver
  A pointer to the *struct uba_driver* structure for this device type.

ui_unit
  The unit number of this device, e.g. 0 in **up0**, or 1 in **dh1**.

ui_ctlr
  The number of the controller on which this device is attached, or −1 if this device is not on a controller.

ui_ubanum
  The number of the UNIBUS on which this device is attached.

ui_slave
  The slave number of this device on the controller which it is attached to, or −1 if the device is not a slave. Thus a disk which was unit 2 on a SC-21 would have *ui_slave* 2; it might or might not be *up2*, that depends on the system configuration specification.

July 27, 1983

ui_intr

> The interrupt vector entries for this device, copied into the UNIBUS interrupt vector at boot time. The values of these fields are filled in by *config* to small code segments which it generates in the file *ubglue.s*.

ui_addr

> The control-status register address of this device.

ui_dk

> The iostat number assigned to this device. Numbers are assigned to disks only, and are small positive integers which index the various *dk_** arrays in *<sys/dk.h>*.

ui_flags

> The optional "flags *xxx*" parameter from the configuration specification was copied to this field, to be interpreted by the driver. If *flags* was not specified, then this field will contain a 0.

ui_alive

> The device is really there. Presently set to 1 when a device is determined to be alive, and left 1.

ui_type

> The device type, to be used by the driver internally.

ui_physaddr

> The physical memory address of the device control-status register. This is used in the device dump routines typically.

ui_mi

> A *struct uba_ctlr* pointer to the controller (if any) on which this device resides.

ui_hd

> A *struct uba_hd* pointer to the UNIBUS on which this device resides.

UNIBUS resource management routines

> UNIBUS drivers are supported by a collection of utility routines which manage UNIBUS resources. If a driver attempts to bypass the UNIBUS routines, other drivers may not operate properly. The major routines are: *uballoc* to allocate UNIBUS resources, *ubarelse* to release previously allocated resources, and *ubago* to initiate DMA. When allocating UNIBUS resources you may request that you

NEEDBDP

> if you need a buffered data path,

HAVEBDP

> if you already have a buffered data path and just want new mapping registers (and access to the UNIBUS), and

CANTWAIT

> if you are calling (potentially) from interrupt level

If the presentation here does not answer all the questions you may have, consult the file /sys/vaxuba/uba.c

Autoconfiguration requirements

> Basically all you have to do is write a *ud_probe* and a *ud_attach* routine for the controller. It suffices to have a *ud_probe* routine which just initializes *br* and *cvec*, and a *ud_attach* routine which does nothing. Making the device fully configurable requires, of course, more work, but is worth it if you expect the device to be in common usage and want to share it with others.

> If you managed to create all the needed hooks, then make sure you include the necessary header files; the ones included by *vaxuba/ct.c* are nearly minimal. Order is important here, don't be surprised at undefined structure complaints if you order the includes wrongly. Finally

um_ctlr

> The controller number for this controller, e.g. the 0 in *sc0*.

um_alive

> Set to 1 if the controller is considered alive; currently, always set for any structure encountered during normal operation. That is, the driver will have a handle on a *uba_ctlr* structure only if the configuration routines set this field to a 1 and entered it into the driver tables.

um_intr

> The interrupt vector routines for this device. These are generated by *config* and this field is initialized in the *ioconf.c* file.

um_hd

> A back-pointer to the UNIBUS adapter to which this controller is attached.

um_cmd

> A place for the driver to store the command which is to be given to the device before calling the routine *ubago* with the devices *uba_device* structure. This information is then retrieved when the device go routine is called and stuffed in the device control status register to start the i/o operation.

um_ubinfo

> Information about the UNIBUS resources allocated to the device. This is normally only used in device driver go routine (as *updgo* above) and occasionally in exceptional condition handling such as ECC correction.

um_tab

> This buffer structure is a place where the driver hangs the device structures which are ready to transfer. Each driver allocates a buf structure for each device (e.g. *updtab* in the *up.c* driver) for this purpose. You can think of this structure as a device-control-block, and the buf structures linked to it as the unit-control-blocks. The code for dealing with this structure is stylized; see the *rk.c* or *up.c* driver for the details. If the *ubago* routine is to be used, the structure attached to this *buf* structure must be:
>
> - A chain of *buf* structures for each waiting device on this controller.
> - On each waiting *buf* structure another *buf* structure which is the one containing the parameters of the i/o operation.

uba_device structure

> One of these structures exist for each device attached to a UNIBUS controller. Devices which are not attached to controllers or which perform no buffered data path DMA i/o may have only a device structure. Thus *dz* and *dh* devices have only *uba_device* structures. The fields are:

ui_driver

> A pointer to the *struct uba_driver* structure for this device type.

ui_unit

> The unit number of this device, e.g. 0 in **up0**, or 1 in **dh1**.

ui_ctlr

> The number of the controller on which this device is attached, or −1 if this device is not on a controller.

ui_ubanum

> The number of the UNIBUS on which this device is attached.

ui_slave

> The slave number of this device on the controller which it is attached to, or −1 if the device is not a slave. Thus a disk which was unit 2 on a SC-21 would have *ui_slave* 2; it might or might not be *up2*, that depends on the system configuration specification.

ui_intr
> The interrupt vector entries for this device, copied into the UNIBUS interrupt vector at boot time. The values of these fields are filled in by *config* to small code segments which it generates in the file *ubglue.s*.

ui_addr
> The control-status register address of this device.

ui_dk
> The iostat number assigned to this device. Numbers are assigned to disks only, and are small positive integers which index the various *dk_** arrays in *<sys/dk.h>*.

ui_flags
> The optional "flags *xxx*" parameter from the configuration specification was copied to this field, to be interpreted by the driver. If *flags* was not specified, then this field will contain a 0.

ui_alive
> The device is really there. Presently set to 1 when a device is determined to be alive, and left 1.

ui_type
> The device type, to be used by the driver internally.

ui_physaddr
> The physical memory address of the device control-status register. This is used in the device dump routines typically.

ui_mi
> A *struct uba_ctlr* pointer to the controller (if any) on which this device resides.

ui_hd
> A *struct uba_hd* pointer to the UNIBUS on which this device resides.

UNIBUS resource management routines

UNIBUS drivers are supported by a collection of utility routines which manage UNIBUS resources. If a driver attempts to bypass the UNIBUS routines, other drivers may not operate properly. The major routines are: *uballoc* to allocate UNIBUS resources, *ubarelse* to release previously allocated resources, and *ubago* to initiate DMA. When allocating UNIBUS resources you may request that you

NEEDBDP
> if you need a buffered data path,

HAVEBDP
> if you already have a buffered data path and just want new mapping registers (and access to the UNIBUS), and

CANTWAIT
> if you are calling (potentially) from interrupt level

If the presentation here does not answer all the questions you may have, consult the file /sys/vaxuba/uba.c

Autoconfiguration requirements

Basically all you have to do is write a *ud_probe* and a *ud_attach* routine for the controller. It suffices to have a *ud_probe* routine which just initializes *br* and *cvec*, and a *ud_attach* routine which does nothing. Making the device fully configurable requires, of course, more work, but is worth it if you expect the device to be in common usage and want to share it with others.

If you managed to create all the needed hooks, then make sure you include the necessary header files; the ones included by *vaxuba/ct.c* are nearly minimal. Order is important here, don't be surprised at undefined structure complaints if you order the includes wrongly. Finally

if you get the device configured in, you can try bootstrapping and see if configuration messages print out about your device. It is a good idea to have some messages in the probe routine so that you can see that you are getting called and what is going on. If you do not get called, then you probably have the control-status register address wrong in your system configuration. The autoconfigure code notices that the device doesn't exist in this case and you will never get called.

Assuming that your probe routine works and you manage to *generate* an interrupt, then you are basically back to where you would have been under older versions of UNIX. Just be sure to use the *ui_ctlr* field of the *uba_device* structures to address the device; compiling in funny constants will make your driver only work on the CPU type you have (780, 750, or 730).

Other bad things that might happen while you are setting up the configuration stuff:

- You get "nexus zero vector" errors from the system. This will happen if you cause a device to interrupt, but take away the interrupt enable so fast that the UNIBUS adapter cancels the interrupt and confuses the processor. The best thing to do it to put a modest delay in the probe code between the instructions which should cause and interrupt and the clearing of the interrupt enable. (You should clear interrupt enable before you leave the probe routine so the device doesn't interrupt more and confuse the system while it is configuring other devices.)

- The device refuses to interrupt or interrupts with a "zero vector". This typically indicates a problem with the hardware or, for devices which emulate other devices, that the emulation is incomplete. Devices may fail to present interrupt vectors because they have configuration switches set wrong, or because they are being accessed in inappropriate ways. Incomplete emulation can cause "maintenance mode" features to not work properly, and these features are often needed to force device interrupts.

6.4. Adding non-standard system facilities

This section considers the work needed to augment *config*'s data base files for non-standard system facilities.

As far as *config* is concerned non-standard facilities may fall into two categories. *Config* understands that certain files are used especially for kernel profiling. These files are indicated in the *files* files with a *profiling-routine* keyword. For example, the current profiling subroutines are sequestered off in a separate file with the following entry:

> *sys/subr_mcount.c*   **optional profiling-routine**

The *profiling-routine* keyword forces *config* to not compile the source file with the −**pg** option.

The second keyword which can be of use is the *config-dependent* keyword. This causes *config* to compile the indicated module with the global configuration parameters. This allows certain modules, such as *machdep.c* to size system data structures based on the maximum number of users configured for the system.

APPENDIX A. CONFIGURATION FILE GRAMMAR

The following grammar is a compressed form of the actual *yacc*(1) grammar used by *config* to parse configuration files. Terminal symbols are shown all in upper case, literals are emboldened; optional clauses are enclosed in brackets, "[" and "]"; zero or more instantiations are denoted with "*".

Configuration ::= [Spec ;]*

Spec ::= Config_spec
      | Device_spec
      | **trace**
      | /* lambda */

/* configuration specifications */

Config_spec ::= **machine** ID
      | **cpu** ID
      | **options** Opt_list
      | **ident** ID
      | System_spec
      | **timezone** [−] NUMBER [**dst** [NUMBER]]
      | **timezone** [−] FPNUMBER [**dst** [NUMBER]]
      | **maxusers** NUMBER

/* system configuration specifications */

System_spec ::= **config** ID System_parameter [System_parameter]*

System_parameter ::= swap_spec | root_spec | dump_spec | arg_spec

swap_spec ::= **swap** [**on**] swap_dev [**and** swap_dev]*

swap_dev ::= dev_spec [**size** NUMBER]

root_spec ::= **root** [**on**] dev_spec

dump_spec ::= **dumps** [**on**] dev_spec

arg_spec ::= **args** [**on**] dev_spec

dev_spec ::= dev_name | major_minor

major_minor ::= **major** NUMBER **minor** NUMBER

dev_name ::= ID [NUMBER [ID]]

/* option specifications */

Opt_list ::= Option [, Option]*

Option ::= ID [= Opt_value]

July 27, 1983

Opt_value ::= ID | NUMBER

/* device specifications */

Device_spec ::= **device** Dev_name Dev_info Int_spec
    | **master** Dev_name Dev_info
    | **disk** Dev_name Dev_info
    | **tape** Dev_name Dev_info
    | **controller** Dev_name Dev_info [Int_spec]
    | **pseudo-device** Dev [NUMBER]

Dev_name ::= Dev NUMBER

Dev ::= **uba** | **mba** | ID

Dev_info ::= Con_info [Info]*

Con_info ::= **at** Dev NUMBER
    | **at nexus** NUMBER

Info ::= **csr** NUMBER
    | **drive** NUMBER
    | **slave** NUMBER
    | **flags** NUMBER

Int_spec ::= **vector** ID [ID]*
    | **priority** NUMBER

Lexical Conventions

The terminal symbols are loosely defined as:

ID

    One or more alphabetics, either upper or lower case, and underscore, "_".

NUMBER

    Approximately the C language specification for an integer number. That is, a leading "0x" indicates a hexadecimal value, a leading "0" indicates an octal value, otherwise the number is expected to be a decimal value. Hexadecimal numbers may use either upper or lower case alphabetics.

FPNUMBER

    A floating point number without exponent. That is a number of the form "nnn.ddd", where the fractional component is optional.

In special instances a question mark, "?", can be substituted for a "NUMBER" token. This is used to effect wildcarding in device interconnection specifications.

Comments in configuration files are indicated by a "#" character at the beginning of the line; the remainder of the line is discarded.

A specification is interpreted as a continuation of the previous line if the first character of the line is tab.

July 27, 1983

APPENDIX B. RULES FOR DEFAULTING SYSTEM DEVICES

When *config* processes a "config" rule which does not fully specify the location of the root file system, paging area(s), device for system dumps, and device for argument list processing it applies a set of rules to define those values left unspecified. The following list of rules are used in defaulting system devices.

1) If a root device is not specified, the swap specification must indicate a "generic" system is to be built.

2) If the root device does not specify a unit number, it defaults to unit 0.

3) If the root device does not include a partition specification, it defaults to the "a" partition.

4) If no swap area is specified, it defaults to the "b" partition of the root device.

5) If no device is specified for processing argument lists, the first swap partition is selected.

6) If no device is chosen for system dumps, the first swap partition is selected (see below to find out where dumps are placed within the partition).

The following table summarizes the default partitions selected when a device specification is incomplete, e.g. "hp0".

Type	Partition
root	"a"
swap	"b"
args	"b"
dumps	"b"

Multiple swap/paging areas

When multiple swap partitions are specified, the system treats the first specified as a "primary" swap area which is always used. The remaining partitions are then interleaved into the paging system at the time a *swapon*(2) system call is made. This is normally done at boot time with a call to *swapon*(8) from the /etc/rc file.

System dumps

System dumps are automatically taken after a system crash, provided the device driver for the "dumps" device supports this. The dump contains the contents of memory, but not the swap areas. Normally the dump device is a disk in which case the information is copied to a location near the back of the partition. The dump is placed in the back of the partition because the primary swap and dump device are commonly the same device and this allows the system to be rebooted without immediately overwriting the saved information. When a dump has occurred, the system variable *dumpsize* is set to a non-zero value indicating the size (in bytes) of the dump. The *savecore* (8) program then copies the information from the dump partition to a file in a "crash" directory and also makes a copy of the system which was running at the time of the crash (usually "/vmunix"). The offset to the system dump is defined in the system variable *dumplo* (a sector offset from the front of the dump partition). The *savecore* program operates by reading the contents of *dumplo*, *dumpdev*, and *dumpmagic* from /dev/kmem, then comparing the value of *dumpmagic* read from /dev/kmem to that located in corresponding location in the dump area of the dump partition. If a match is found, *savecore* assumes a crash occurred and reads *dumpsize* from the dump area of the dump partition. This value is then used in copying the system dump. Refer to *savecore* (8) for more information about its operation.

The value *dumplo* is calculated to be

dumpdev-size − DUMPDEV

July 27, 1983

where *dumpdev-size* is the size of the disk partition where system dumps are to be placed, and DUMPDEV is 10 Megabytes. If the disk partition is not large enough to hold a 10 Megabyte dump, *dumplo* is set to 0 (the front of the partition). For sites with more than 10 Megabytes of memory the definition of DUMPDEV in /sys/vax/autoconf.c will have to be changed.

APPENDIX C. SAMPLE CONFIGURATION FILES

The following configuration files are developed in section 5; they are included here for completeness.

```
#
# ANSEL VAX (a picture perfect machine)
#
machine          vax
cpu              VAX780
timezone         8 dst
ident            ANSEL
maxusers         40

config           vmunix        root on hp0
config           hpvmunix      root on hp0 swap on hp0 and hp2
config           genvmunix     swap generic

controller       mba0          at nexus ?
disk             hp0           at mba? disk ?
disk             hp1           at mba? disk ?
controller       mba1          at nexus ?
disk             hp2           at mba? disk ?
disk             hp3           at mba? disk ?
controller       uba0          at nexus ?
controller       tm0           at uba? csr 0172520    vector tmintr
tape             te0           at tm0 drive 0
tape             te1           at tm0 drive 1
device           dh0           at uba? csr 0160020    vector dhrint dhxint
device           dm0           at uba? csr 0170500    vector dmintr
device           dh1           at uba? csr 0160040    vector dhrint dhxint
device           dh2           at uba? csr 0160060    vector dhrint dhxint
```

July 27, 1983

```
#
# UCBVAX - Gateway to the world
#
machine          vax
cpu              "VAX780"
cpu              "VAX750"
ident            UCBVAX
timezone         8 dst
maxusers         32
options          INET

config           vmunix        root on hp swap on hp and rk0 and rk1
config           upvmunix      root on up
config           hkvmunix      root on hk swap on rk0 and rk1

controller       mba0          at nexus ?
controller       uba0          at nexus ?
disk             hp0           at mba? drive 0
disk             hp1           at mba? drive 1
controller       sc0           at uba? csr 0176700     vector upintr
disk             up0           at sc0 drive 0
disk             up1           at sc0 drive 1
controller       hk0           at uba? csr 0177440     vector rkintr
disk             rk0           at hk0 drive 0
disk             rk1           at hk0 drive 1
pseudo-device    inet
pseudo-device    pty
# software loopback device for testing
pseudo-device    loop
pseudo-device    imp
device           acc0          at uba? csr 0167600     vector accrint accxint
pseudo-device    ether
device           ec0           at uba? csr 0164330     vector ecrint eccollide ecxint
device           il0           at uba? csr 0164000     vector ilrint ilcint
```

July 27, 1983

APPENDIX D. VAX KERNEL DATA STRUCTURE SIZING RULES

Certain system data structures are sized at compile time according to the maximum number of simultaneous users expected, while others are calculated at boot time based on the physical resources present; e.g. memory. This appendix lists both sets of rules and also includes some hints on changing built-in limitations on certain data structures.

Compile time rules

The file */sys/conf/param.c* contains the definitions of almost all data structures sized at compile time. This file is copied into the directory of each configured system to allow configuration-dependent rules and values to be maintained. The rules implied by its contents are summarized below (here MAXUSERS refers to the value defined in the configuration file in the "maxusers" rule).

nproc

The maximum number of processes which may be running at any time. It is defined to be 20 + 8 * MAXUSERS and referred to in other calculations as NPROC.

ntext

The maximum number of active shared text segments. Defined as 24 + MAXUSERS + NETSLOP, where NETSLOP is 20 when the Internet protocols are configured in the system and 0 otherwise. The added size for supporting the network is to take into account the numerous server processes which are likely to exist.

ninode

The maximum number of files in the file system which may be active at any time. This includes files in use by users, as well as directory files being read or written by the system and files associated with bound sockets in the UNIX ipc domain. This is defined as (NPROC + 16 + MAXUSERS) + 32.

nfile

The number of "file table" structures. One file table structure is used for each open, unshared, file descriptor. Multiple file descriptors may reference a single file table entry when they are created through a *dup* call, or as the result of a *fork*. This is defined to be

$$16 * (NPROC + 16 + MAXUSERS) / 10 + 32 + 2 * NETSLOP$$

where NETSLOP is defined as for **ntext**.

ncallout

The number of "callout" structures. One callout structure is used per internal system event handled with a timeout. Timeouts are used for terminal delays, watchdog routines in device drivers, protocol timeout processing, etc. This is defined as 16 + NPROC.

nclist

The number of "c-list" structures. C-list structures are used in terminal i/o. This is defined as 100 + 16 * MAXUSERS.

nmbclusters

The maximum number of pages which may be allocated by the network. This is defined as 256 (a quarter megabyte of memory) in /sys/h/mbuf.h. In practice, the network rarely uses this much memory. It starts off by allocating 64 kilobytes of memory, then requesting more as required. This value represents an upper bound.

nquota

The number of "quota" structures allocated. Quota structures are present only when disc quotas are configured in the system. One quota structure is kept per user. This is defined to be (MAXUSERS * 9) / 7 + 3.

ndquot

> The number of "dquot" structures allocated. Dquot structures are present only when disc quotas are configured in the system. One dquot structure is required per user, per active file system quota. That is, when a user manipulates a file on a file system on which quotas are enabled, the information regarding the user's quotas on that file system must be in-core. This information is cached, so that not all information must be present in-core all the time. This is defined as (MAXUSERS * NMOUNT) / 4 + NPROC, where NMOUNT is the maximum number of mountable file systems.

In addition to the above values, the system page tables (used to map virtual memory in the kernel's address space) are sized at compile time by the SYSPTSIZE definition in the file /sys/vax/vmparam.h. This is defined to be 20 + MAXUSERS pages of page tables. Its definition affects the size of many data structures allocated at boot time because it constrains the amount of virtual memory which may be addressed by the running system. This is often the limiting factor in the size of the buffer cache.

Run-time calculations

The most important data structures sized at run-time are those used in the buffer cache. Allocation is done by swiping physical memory (and the associated virtual memory) immediately after the system has been started up; look in the file /sys/vax/machdep.c. The amount of physical memory which may be allocated to the buffer cache is constrained by the size of the system page tables, among other things. While the system may calculate a large amount of memory to be allocated to the buffer cache, if the system page table is too small to map this physical memory into the virtual address space of the system, only as much as can be mapped will be used.

The buffer cache is comprised of a number of "buffer headers" and a pool of pages attached to these headers. Buffer headers are divided into two categories: those used for swapping and paging, and those used for normal file i/o. The system tries to allocate 10% of available physical memory for the buffer cache (where *available* does not count that space occupied by the system's text and data segments). If this results in fewer than 16 pages of memory allocated, then 16 pages are allocated. This value is kept in the initialized variable *bufpages* so that it may be patched in the binary image (to allow tuning without recompiling the system). A sufficient number of file i/o buffer headers are then allocated to allow each to hold 2 pages each, and half as many swap i/o buffer headers are then allocated. The number of swap i/o buffer headers is constrained to be no more than 256.

System size limitations

As distributed, the sum of the virtual sizes of the core-resident processes is limited to 64M bytes. The size of the text, and data segments of a single process are currently limited to 6M bytes each, and the stack segment size is limited to 512K bytes as a soft, user-changeable limit, and may be increased to 6M with the *setrlimit* (2) system call. If these are insufficient, they can be increased by changing the constants MAXTSIZ, MAXDSIZ and MAXSSIZ in the file /sys/vax/vmparam.h. The size of the swap maps for these objects must also be increased; for text, the parameters are NXDAD (/sys/h/text.h) and DMTEXT (/sys/vax/autoconfig.c). The maps for data and swap are limited by NDMAP (/sys/h/dmap.h) and DMMAX (/sys/vax/autoconfig.c). You must be careful in doing this that you have adequate paging space. As normally configured , the system has only 16M bytes per paging area. The best way to get more space is to provide multiple, thereby interleaved, paging areas.

To increase the amount of resident virtual space possible, you can alter the constant USRPTSIZE (in /sys/vax/vmparam.h). To allow 128 megabytes of resident virtual space one would change the 8 to a 16.

Because the file system block numbers are stored in page table *pg_blkno* entries, the maximum size of a file system is limited to 2^19 1024 byte blocks. Thus no file system can be larger than 512M bytes.

July 27, 1983

The count of mountable file systems is limited to 15. This should be sufficient. If you have many disks it makes sense to make some of them single file systems, and the paging areas don't count in this total. To increase this it will be necessary to change the core-map /sys/h/cmap.h since there is a 4 bit field used here. The size of the core-map will then expand to 16 bytes per 1024 byte page. (Don't forget to change MSWAPX and NMOUNT in /sys/h/param.h also.)

The maximum value NOFILE (open files per process limit) can be raised to is 30 because of a bit field in the page table entry in /sys/machine/pte.h.

The amount of physical memory is currently limited to 8 Mb by the size of the index fields in the core-map (/sys/h/cmap.h). This limit is also found in /sys/vax/locore.s.

July 27, 1983

Disc Quotas in a UNIX* Environment.

Robert Elz

Department of Computer Science
University of Melbourne,
Parkville,
Victoria,
Australia.

ABSTRACT

In most computing environments, disc space is not infinite. The disc quota system provides a mechanism to control usage of disc space, on an individual basis.

Quotas may be set for each individual user, on any, or all filesystems.

The quota system will warn users when they exceed their allotted limit, but allow some extra space for current work. Repeatedly remaining over quota at logout, will cause a fatal over quota condition eventually.

The quota system is an optional part of VMUNIX that may be included when the system is configured.

5th July, 1983

---

* UNIX is a trademark of Bell Laboratories.

Disc Quotas in a UNIX* Environment.

Robert Elz

Department of Computer Science
University of Melbourne,
Parkville,
Victoria,
Australia.

1. Users' view of disc quotas

To most users, disc quotas will either be of no concern, or a fact of life that cannot be avoided. The *quota* (1) command will provide information on any disc quotas that may have been imposed upon a user.

There are two individual possible quotas that may be imposed, usually if one is, both will be. A limit can be set on the amount of space a user can occupy, and there may be a limit on the number of files (inodes) he can own.

Quota provides information on the quotas that have been set by the system administrators, in each of these areas, and current usage.

There are four numbers for each limit, the current usage, soft limit (quota), hard limit, and number of remaining login warnings. The soft limit is the number of 1K blocks (or files) that the user is expected to remain below. Each time the user's usage goes past this limit, he will be warned. The hard limit cannot be exceeded. If a user's usage reaches this number, further requests for space (or attempts to create a file) will fail with an EDQUOT error, and the first time this occurs, a message will be written to the user's terminal. Only one message will be output, until space occupied is reduced below the limit, and reaches it again, in order to avoid continual noise from those programs that ignore write errors.

Whenever a user logs in with a usage greater than his soft limit, he will be warned, and his login warning count decremented. When he logs in under quota, the counter is reset to its maximum value (which is a *system* configuration parameter, that is typically 3). If the warning count should ever reach zero (caused by three successive logins over quota), the particular limit that has been exceeded will be treated as if the hard limit has been reached, and no more resources will be allocated to the user. The **only** way to reset this condition is to reduce usage below quota, then log in again.

1.1. Surviving when quota limit is reached

In most cases, the only way to recover from over quota conditions, is to abort whatever activity was in progress on the filesystem that has reached its limit, remove sufficient files to bring the limit back below quota, and retry the failed program.

However, if you are in the editor and a write fails because of an over quota situation, that is not a suitable course of action, as it is most likely that initially attempting to write the file will have truncated its previous contents, so should the editor be aborted without correctly writing the file not only will the recent changes be lost, but possibly much, or even all, of the data that previously existed.

There are several possible safe exits for a user caught in this situation. He may use the editor ! shell escape command to examine his file space, and remove surplus files.

---

* UNIX is a trademark of Bell Laboratories.

Alternatively, using *csh*, he may suspend the editor, remove some files, then resume it. A third possibility, is to write the file to some other filesystem (perhaps to a file on /tmp) where the user's quota has not been exceeded. Then after rectifying the quota situation, the file can be moved back to the filesystem it belongs on.

2. Administering the quota system

To set up and establish the disc quota system, there are several steps necessary to be performed by the system administrator.

First, the system must be configured to include the disc quota sub-system. This is done by including the line:

options   QUOTA

in the system configuration file, then running *config* (8) followed by a system configuration*.

Second, a decision as to what filesystems need to have quotas applied needs to be made. Usually, only filesystems that house users' home directories, or other user files, will need to be subjected to the quota system, though it may also prove useful to also include **/usr**. If possible, **/tmp** should usually be free of quotas.

Having decided on which filesystems quotas need to be set upon, the administrator should then allocate the available space amongst the competing needs. How this should be done is (way) beyond the scope of this document.

Then, the *edquota* (8) command can be used to actually set the limits desired upon each user. Where a number of users are to be given the same quotas (a common occurrence) the **−p** switch to edquota will allow this to be easily accomplished.

Once the quotas are set, ready to operate, the system must be informed to enforce quotas on the desired filesystems. This is accomplished with the *quotaon* (8) command. *Quotaon* will either enable quotas for a particular filesystem, or with the **−a** switch, will enable quotas for each filesystem indicated in **/etc/fstab** as using quotas. See *fstab* (5) for details. Most sites using the quota system, will include the line

/etc/quotaon -a

in **/etc/rc.local**.

Should quotas need to be disabled, the *quotaoff*(8) command will do that, however, should the filesystem be about to be dismounted, the *umount* (8) command will disable quotas immediately before the filesystem is unmounted. This is actually an effect of the *umount* (2) system call, and it guarantees that the quota system will not be disabled if the umount would fail because the filesystem is not idle.

Periodically (certainly after each reboot, and when quotas are first enabled for a filesystem), the records retained in the quota file should be checked for consistency with the actual number of blocks and files allocated to the user. The *quotachk* (8) command can be used to accomplish this. It is not necessary to dismount the filesystem, or disable the quota system to run this command, though on active filesystems inaccurate results may occur. This does no real harm in most cases, another run of *quotachk* when the filesystem is idle will certainly correct any inaccuracy.

The super-user may use the *quota* (1) command to examine the usage and quotas of any user, and the *repquota* (8) command may be used to check the usages and limits for all users on a filesystem.

---

* See also the document "Building 4.2BSD UNIX Systems with Config".

3. Some implementation detail.

Disc quota usage and information is stored in a file on the filesystem that the quotas are to be applied to. Conventionally, this file is **quotas** in the root of the filesystem. While this name is not known to the system in any way, several of the user level utilities "know" it, and choosing any other name would not be wise.

The data in the file comprises an array of structures, indexed by uid, one structure for each user on the system (whether the user has a quota on this filesystem or not). If the uid space is sparse, then the file may have holes in it, which would be lost by copying, so it is best to avoid this.

The system is informed of the existence of the quota file by the *setquota* (2) system call. It then reads the quota entries for each user currently active, then for any files open owned by users who are not currently active. Each subsequent open of a file on the filesystem, will be accompanied by a pairing with its quota information. In most cases this information will be retained in core, either because the user who owns the file is running some process, because other files are open owned by the same user, or because some file (perhaps this one) was recently accessed. In memory, the quota information is kept hashed by user-id and filesystem, and retained in an LRU chain so recently released data can be easily reclaimed. Information about those users whose last process has recently terminated is also retained in this way.

Each time a block is accessed or released, and each time an inode is allocated or freed, the quota system gets told about it, and in the case of allocations, gets the opportunity to object.

Measurements have shown that the quota code uses a very small percentage of the system cpu time consumed in writing a new block to disc.

4. Acknowledgments

The current disc quota system is loosely based upon a very early scheme implemented at the University of New South Wales, and Sydney University in the mid 70's. That system implemented a single combined limit for both files and blocks on all filesystems.

A later system was implemented at the University of Melbourne by the author, but was not kept highly accurately, eg: chown's (etc) did not affect quotas, nor did i/o to a file other than one owned by the instigator.

The current system has been running (with only minor modifications) since January 82 at Melbourne. It is actually just a small part of a much broader resource control scheme, which is capable of controlling almost anything that is usually uncontrolled in unix. The rest of this is, as yet, still in a state where it is far too subject to change to be considered for distribution.

For the 4.2BSD release, much work has been done to clean up and sanely incorporate the quota code by Sam Leffler and Kirk McKusick at The University of California at Berkeley.

Fsck − The UNIX† File System Check Program

Revised July 28, 1983

Marshall Kirk McKusick

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

T. J. Kowalski

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This document reflects the use of *fsck* with the 4.2BSD file system organization. This is a revision of the original paper written by T. J. Kowalski.

File System Check Program (*fsck*) is an interactive file system check and repair program. *Fsck* uses the redundant structural information in the UNIX file system to perform several consistency checks. If an inconsistency is detected, it is reported to the operator, who may elect to fix or ignore each inconsistency. These inconsistencies result from the permanent interruption of the file system updates, which are performed every time a file is modified. Unless there has been a hardware failure, *fsck* is able to repair corrupted file systems using procedures based upon the order in which UNIX honors these file system update requests.

The purpose of this document is to describe the normal updating of the file system, to discuss the possible causes of file system corruption, and to present the corrective actions implemented by *fsck*. Both the program and the interaction between the program and the operator are described.

TABLE OF CONTENTS

1. Introduction

This document reflects the use of *fsck* with the 4.2BSD file system organization. This is a revision of the original paper written by T. J. Kowalski.

When a UNIX operating system is brought up, a consistency check of the file systems should always be performed. This precautionary measure helps to insure a reliable environment for file storage on disk. If an inconsistency is discovered, corrective action must be taken. *Fsck* runs in two modes. Normally it is run non-interactively by the system after a normal boot. When running in this mode, it will only make changes to the file system that are known to always be correct. If an unexpected inconsistency is found *fsck* will exit with a non-zero exit status, leaving the system running single-user. Typically the operator then runs *fsck* interactively. When running in this mode, each problem is listed followed by a suggested corrective action. The operator must decide whether or not the suggested correction should be made.

The purpose of this memo is to dispel the mystique surrounding file system inconsistencies. It first describes the updating of the file system (the calm before the storm) and then describes file system corruption (the storm). Finally, the set of deterministic corrective actions used by *fsck* (the Coast Guard to the rescue) is presented.

2. Overview of the file system

The file system is discussed in detail in [Mckusick83]; this section gives a brief overview.

2.1. Superblock

A file system is described by its *super-block*. The super-block is built when the file system is created (*newfs*(8)) and never changes. The super-block contains the basic parameters of the file system, such as the number of data blocks it contains and a count of the maximum number of files. Because the super-block contains critical data, *newfs* replicates it to protect against catastrophic loss. The *default super block* always resides at a fixed offset from the beginning of the file system's disk partition. The *redundant super blocks* are not referenced unless a head crash or other hard disk error causes the default super-block to be unusable. The redundant blocks are sprinkled throughout the disk partition.

Within the file system are files. Certain files are distinguished as directories and contain collections of pointers to files that may themselves be directories. Every file has a descriptor associated with it called an *inode*. The inode contains information describing ownership of the file, time stamps indicating modification and access times for the file, and an array of indices pointing to the data blocks for the file. In this section, we assume that the first 12 blocks of the file are directly referenced by values stored in the inode structure itself†. The inode structure may also contain references to indirect blocks containing further data block indices. In a file system with a 4096 byte block size, a singly indirect block contains 1024 further block addresses, a doubly indirect block contains 1024 addresses of further single indirect blocks, and a triply indirect block contains 1024 addresses of further doubly indirect blocks.

In order to create files with up to $2\uparrow32$ bytes, using only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of two greater than or equal to 4096. The block size of the file system is maintained in the super-block, so it is possible for file systems of different block sizes to be accessible simultaneously on the same system. The block size must be decided when *newfs* creates the file system; the block size cannot be subsequently changed without rebuilding the file system.

2.2. Summary information

Associated with the super block is non replicated *summary information*. The summary information changes as the file system is modified. The summary information contains the number of blocks, fragments, inodes and directories in the file system.

2.3. Cylinder groups

The file system partitions the disk into one or more areas called *cylinder groups*. A cylinder group is comprised of one or more consecutive cylinders on a disk. Each cylinder group includes inode slots for files, a *block map* describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. A fixed number of inodes is allocated for each cylinder group when the file system is created. The current policy is to allocate one inode for each 2048 bytes of disk space; this is expected to be far more inodes than will ever be needed.

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all the redundant information would be on the top platter. A single hardware failure that destroyed the top platter could cause the loss of all copies of the redundant super-blocks. Thus the cylinder group bookkeeping information begins at a floating offset from the beginning of the cylinder group. The offset for the $i+1$st cylinder group is about one track further from the beginning of the cylinder group than it was for the ith cylinder group. In this way, the redundant information spirals down into the pack; any single track, cylinder, or platter can be lost without losing all copies of the super-blocks.

---

†The actual number may vary from system to system, but is usually in the range 5-13.

Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information stores data.

2.4. Fragments

To avoid waste in storing small files, the file system space allocator divides a single file system block into one or more *fragments*. The fragmentation of the file system is specified when the file system is created; each file system block can be optionally broken into 2, 4, or 8 addressable fragments. The lower bound on the size of these fragments is constrained by the disk sector size; typically 512 bytes is the lower bound on fragment size. The block map associated with each cylinder group records the space availability at the fragment level. Aligned fragments are examined to determine block availability.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096 byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remainder of the block is made available for allocation to other files. For example, consider an 11000 byte file stored on a 4096/1024 byte file system. This file uses two full size blocks and a 3072 byte fragment. If no fragments with at least 3072 bytes are available when the file is created, a full size block is split yielding the necessary 3072 byte fragment and an unused 1024 byte fragment. This remaining fragment can be allocated to another file, as needed.

2.5. Updates to the file system

Every working day hundreds of files are created, modified, and removed. Every time a file is modified, the operating system performs a series of file system updates. These updates, when written on disk, yield a consistent file system. The file system stages all modifications of critical information; modification can either be completed or cleanly backed out after a crash. Knowing the information that is first written to the file system, deterministic procedures can be developed to repair a corrupted file system. To understand this process, the order that the update requests were being honored must first be understood.

When a user program does an operation to change the file system, such as a *write*, the data to be written is copied into an internal *in-core* buffer in the kernel. Normally, the disk update is handled asynchronously; the user process is allowed to proceed even though the data has not yet been written to the disk. The data, along with the inode information reflecting the change, is eventually written out to disk. The real disk write may not happen until long after the *write* system call has returned. Thus at any given time, the file system, as it resides on the disk, lags the state of the file system represented by the in-core information.

The disk information is updated to reflect the in-core information when the buffer is required for another use, when a *sync* (2) is done (at 30 second intervals) by */etc/update* (8), or by manual operator intervention with the *sync* (8) command. If the system is halted without writing out the in-core information, the file system on the disk will be in an inconsistent state.

If all updates are done asynchronously, several serious inconsistencies can arise. One inconsistency is that a block may be claimed by two inodes. Such an inconsistency can occur when the system is halted before the pointer to the block in the old inode has been cleared in the copy of the old inode on the disk, and after the pointer to the block in the new inode has been written out to the copy of the new inode on the disk. Here, there is no deterministic method for deciding which inode should really claim the block. A similar problem can arise with a multiply claimed inode.

The problem with asynchronous inode updates can be avoided by doing all inode deallocations synchronously. Consequently, inodes and indirect blocks are written to the disk synchronously (*i.e.* the process blocks until the information is really written to disk) when they are being deallocated. Similarly inodes are kept consistent by synchronously deleting, adding, or changing directory entries.

3. Fixing corrupted file systems

A file system can become corrupted in several ways. The most common of these ways are improper shutdown procedures and hardware failures.

File systems may become corrupted during an *unclean halt*. This happens when proper shutdown procedures are not observed, physically write-protecting a mounted file system, or a mounted file system is taken off-line. The most common operator procedural failure is forgetting to *sync* the system before halting the CPU.

File systems may become further corrupted if proper startup procedures are not observed, e.g., not checking a file system for inconsistencies, and not repairing inconsistencies. Allowing a corrupted file system to be used (and, thus, to be modified further) can be disastrous.

Any piece of hardware can fail at any time. Failures can be as subtle as a bad block on a disk pack, or as blatant as a non-functional disk-controller.

3.1. Detecting and correcting corruption

Normally *fsck* is run non-interactively. In this mode it will only fix corruptions that are expected to occur from an unclean halt. These actions are a proper subset of the actions that *fsck* will take when it is running interactively. Throughout this paper we assume that *fsck* is being run interactively, and all possible errors can be encountered. When an inconsistency is discovered in this mode, *fsck* reports the inconsistency for the operator to chose a corrective action.

A quiescent‡ file system may be checked for structural integrity by performing consistency checks on the redundant data intrinsic to a file system. The redundant data is either read from the file system, or computed from other known values. The file system **must** be in a quiescent state when *fsck* is run, since *fsck* is a multi-pass program.

In the following sections, we discuss methods to discover inconsistencies and possible corrective actions for the cylinder group blocks, the inodes, the indirect blocks, and the data blocks containing directory entries.

3.2. Super-block checking

The most commonly corrupted item in a file system is the summary information associated with the super-block. The summary information is prone to corruption because it is modified with every change to the file system's blocks or inodes, and is usually corrupted after an unclean halt.

The super-block is checked for inconsistencies involving file-system size, number of inodes, free-block count, and the free-inode count. The file-system size must be larger than the number of blocks used by the super-block and the number of blocks used by the list of inodes. The file-system size and layout information are the most critical pieces of information for *fsck*. While there is no way to actually check these sizes, since they are statically determined by *newfs*, *fsck* can check that these sizes are within reasonable bounds. All other file system checks require that these sizes be correct. If *fsck* detects corruption in the static parameters of the default super-block, *fsck* requests the operator to specify the location of an alternate super-block.

3.3. Free block checking

Fsck checks that all the blocks marked as free in the cylinder group block maps are not claimed by any files. When all the blocks have been initially accounted for, *fsck* checks that the number of free blocks plus the number of blocks claimed by the inodes equals the total number of blocks in the file system.

---

‡ I.e., unmounted and not being written on.

If anything is wrong with the block allocation maps, *fsck* will rebuild them, based on the list it has computed of allocated blocks.

The summary information associated with the super-block counts the total number of free blocks within the file system. *Fsck* compares this count to the number of free blocks it found within the file system. If the two counts do not agree, then *fsck* replaces the incorrect count in the summary information by the actual free-block count.

The summary information counts the total number of free inodes within the file system. *Fsck* compares this count to the number of free inodes it found within the file system. If the two counts do not agree, then *fsck* replaces the incorrect count in the summary information by the actual free-inode count.

3.4. Checking the inode state

An individual inode is not as likely to be corrupted as the allocation information. However, because of the great number of active inodes, a few of the inodes are usually corrupted.

The list of inodes in the file system is checked sequentially starting with inode 2 (inode 0 marks unused inodes; inode 1 is saved for future generations) and progressing through the last inode in the file system. The state of each inode is checked for inconsistencies involving format and type, link count, duplicate blocks, bad blocks, and inode size.

Each inode contains a mode word. This mode word describes the type and state of the inode. Inodes must be one of six types: regular inode, directory inode, symbolic link inode, special block inode, special character inode, or socket inode. Inodes may be found in one of three allocation states: unallocated, allocated, and neither unallocated nor allocated. This last state suggests an incorrectly formated inode. An inode can get in this state if bad data is written into the inode list. The only possible corrective action is for *fsck* is to clear the inode.

3.5. Inode links

Each inode counts the total number of directory entries linked to the inode. *Fsck* verifies the link count of each inode by starting at the root of the file system, and descending through the directory structure. The actual link count for each inode is calculated during the descent.

If the stored link count is non-zero and the actual link count is zero, then no directory entry appears for the inode. If this happens, *fsck* will place the disconnected file in the *lost+found* directory. If the stored and actual link counts are non-zero and unequal, a directory entry may have been added or removed without the inode being updated. If this happens, *fsck* replaces the incorrect stored link count by the actual link count.

Each inode contains a list, or pointers to lists (indirect blocks), of all the blocks claimed by the inode. Since indirect blocks are owned by an inode, inconsistencies in indirect blocks directly affect the inode that owns it.

Fsck compares each block number claimed by an inode against a list of already allocated blocks. If another inode already claims a block number, then the block number is added to a list of *duplicate blocks*. Otherwise, the list of allocated blocks is updated to include the block number.

If there are any duplicate blocks, *fsck* will perform a partial second pass over the inode list to find the inode of the duplicated block. The second pass is needed, since without examining the files associated with these inodes for correct content, not enough information is available to determine which inode is corrupted and should be cleared. If this condition does arise (only hardware failure will cause it), then the inode with the earliest modify time is usually incorrect, and should be cleared. If this happens, *fsck* prompts the operator to clear both inodes. The operator must decide which one should be kept and which one should be cleared.

Fsck checks the range of each block number claimed by an inode. If the block number is lower than the first data block in the file system, or greater than the last data block, then the block number is a *bad block number*. Many bad blocks in an inode are usually caused by an

indirect block was not written to the file system, a condition which can only occur if there has been a hardware failure. If an inode contains bad block numbers, *fsck* prompts the operator to clear it.

3.6. Inode data size

Each inode contains a count of the number of data blocks that it contains. The number of actual data blocks is the sum of the allocated data blocks and the indirect blocks. *Fsck* computes the actual number of data blocks and compares that block count against the actual number of blocks the inode claims. If an inode contains an incorrect count *fsck* prompts the operator to fix it.

Each inode contains a thirty-two bit size field. The size is the number of data bytes in the file associated with the inode. The consistency of the byte size field is roughly checked by computing from the size field the maximum number of blocks that should be associated with the inode, and comparing that expected block count against the actual number of blocks the inode claims.

3.7. Checking the data associated with an inode

An inode can directly or indirectly reference three kinds of data blocks. All referenced blocks must be the same kind. The three types of data blocks are: plain data blocks, symbolic link data blocks, and directory data blocks. Plain data blocks and symbolic link data blocks contain the information stored in a file. Directory data blocks contain directory entries. *Fsck* can only check the validity of directory data blocks.

Each directory data block is checked for several types of inconsistencies. These inconsistencies include directory inode numbers pointing to unallocated inodes, directory inode numbers that are greater than the number of inodes in the file system, incorrect directory inode numbers for "." and "..", and directories that are not attached to the file system. If the inode number in a directory data block references an unallocated inode, then *fsck* will remove that directory entry. Again, this condition can only arise when there has been a hardware failure.

If a directory entry inode number references outside the inode list, then *fsck* will remove that directory entry. This condition occurs if bad data is written into a directory data block.

The directory inode number entry for "." must be the first entry in the directory data block. The inode number for "." must reference itself; e.g., it must equal the inode number for the directory data block. The directory inode number entry for ".." must be the second entry in the directory data block. Its value must equal the inode number for the parent of the directory entry (or the inode number of the directory data block if the directory is the root directory). If the directory inode numbers are incorrect, *fsck* will replace them with the correct values.

3.8. File system connectivity

Fsck checks the general connectivity of the file system. If directories are not linked into the file system, then *fsck* links the directory back into the file system in the *lost+found* directory. This condition only occurs when there has been a hardware failure.

Acknowledgements

References

[Dolotta78]   Dolotta, T. A., and Olsson, S. B. eds., *UNIX User's Manual, Edition 1.1* (January 1978).

[Joy83]   Joy, W., Cooper, E., Fabry, R., Leffler, S., McKusick, M., and Mosher, D. *4.2BSD System Manual*, University of California at Berkeley, Computer Systems Research Group Technical Report #4, 1982.

[McKusick83]   McKusick, M., Joy, W., Leffler, S., and Fabry, R. *A Fast File System for UNIX*, University of California at Berkeley, Computer Systems Research Group Technical Report #7, 1982.

[Ritchie78]   Ritchie, D. M., and Thompson, K., The UNIX Time-Sharing System, *The Bell System Technical Journal* 57, 6 (July-August 1978, Part 2), pp. 1905-29.

[Thompson78]   Thompson, K., UNIX Implementation, *The Bell System Technical Journal* 57, 6 (July-August 1978, Part 2), pp. 1931-46.

4. Appendix A — Fsck Error Conditions

4.1. Conventions

Fsck is a multi-pass file system check program. Each file system pass invokes a different Phase of the *fsck* program. After the initial setup, *fsck* performs successive Phases over each file system, checking blocks and sizes, path-names, connectivity, reference counts, and the map of free blocks, (possibly rebuilding it), and performs some cleanup.

Normally *fsck* is run non-interactively to *preen* the file systems after an unclean halt. While preen'ing a file system, it will only fix corruptions that are expected to occur from an unclean halt. These actions are a proper subset of the actions that *fsck* will take when it is running interactively. Throughout this appendix many errors have several options that the operator can take. When an inconsistency is detected, *fsck* reports the error condition to the operator. If a response is required, *fsck* prints a prompt message and waits for a response. When preen'ing most errors are fatal. For those that are expected, the response taken is noted. This appendix explains the meaning of each error condition, the possible responses, and the related error conditions.

The error conditions are organized by the *Phase* of the *fsck* program in which they can occur. The error conditions that may occur in more than one Phase will be discussed in initialization.

4.2. Initialization

Before a file system check can be performed, certain tables have to be set up and certain files opened. This section concerns itself with the opening of files and the initialization of tables. This section lists error conditions resulting from command line options, memory requests, opening of files, status of files, file system size checks, and creation of the scratch file. All of the initialization errors are fatal when the file system is being preen'ed.

C option?
C is not a legal option to *fsck*; legal options are −b, −y, −n, and −p. *Fsck* terminates on this error condition. See the *fsck*(8) manual entry for further detail.

cannot alloc NNN bytes for blockmap
cannot alloc NNN bytes for freemap
cannot alloc NNN bytes for statemap
cannot alloc NNN bytes for lncntp
Fsck's request for memory for its virtual memory tables failed. This should never happen. *Fsck* terminates on this error condition. See a guru.

Can't open checklist file: *F*
The file system checklist file *F* (usually */etc/fstab*) can not be opened for reading. *Fsck* terminates on this error condition. Check access modes of *F*.

Can't stat root
Fsck's request for statistics about the root directory "/" failed. This should never happen. *Fsck* terminates on this error condition. See a guru.

Can't stat *F*
Can't make sense out of name *F*
Fsck's request for statistics about the file system *F* failed. When running manually, it ignores this file system and continues checking the next file system given. Check access modes of *F*.

Can't open *F*
Fsck's request attempt to open the file system *F* failed. When running manually, it ignores this

file system and continues checking the next file system given. Check access modes of *F*.

F: (NO WRITE)
Either the −n flag was specified or *fsck*'s attempt to open the file system *F* for writing failed.
When running manually, all the diagnostics are printed out, but no modifications are attempted
to fix them.

file is not a block or character device; OK
You have given *fsck* a regular file name by mistake. Check the type of the file specified.

Possible responses to the OK prompt are:

YES Ignore this error condition.

NO   ignore this file system and continues checking the next file system given.

One of the following messages will appear:
MAGIC NUMBER WRONG
NCG OUT OF RANGE
CPG OUT OF RANGE
NCYL DOES NOT JIVE WITH NCG*CPG
SIZE PREPOSTEROUSLY LARGE
TRASHED VALUES IN SUPER BLOCK

and will be followed by the message:
F: BAD SUPER BLOCK: *B*
**USE -b OPTION TO FSCK TO SPECIFY LOCATION OF AN ALTERNATE
SUPER-BLOCK TO SUPPLY NEEDED INFORMATION; SEE fsck(8).**
The super block has been corrupted. An alternative super block must be selected from among
those listed by *newfs* (8) when the file system was created. For file systems with a blocksize
less than 32K, specifying −b 32 is a good first choice.

INTERNAL INCONSISTENCY: *M*
Fsck's has had an internal panic, whose message is specified as *M*. This should never happen.
See a guru.

CAN NOT SEEK: BLK *B* (CONTINUE)
Fsck's request for moving to a specified block number *B* in the file system failed. This should
never happen. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however the problem will persist.
    This error condition will not allow a complete check of the file system. A second run of
    fsck should be made to re-check this file system. If the block was part of the virtual
    memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO   terminate the program.

CAN NOT READ: BLK *B* (CONTINUE)
Fsck's request for reading a specified block number *B* in the file system failed. This should
never happen. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however, the problem will per-
    sist. This error condition will not allow a complete check of the file system. A second
    run of *fsck* should be made to re-check this file system. If the block was part of the

virtual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO   terminate the program.

CAN NOT WRITE: BLK *B* (CONTINUE)
Fsck's request for writing a specified block number *B* in the file system failed. The disk is write-protected. See a guru.

Possible responses to the CONTINUE prompt are:

YES attempt to continue to run the file system check. Often, however, the problem will per-
     sist. This error condition will not allow a complete check of the file system. A second
     run of *fsck* should be made to re-check this file system. If the block was part of the vir-
     tual memory buffer cache, *fsck* will terminate with the message "Fatal I/O error".

NO   terminate the program.

4.3. Phase 1 — Check Blocks and Sizes
     This phase concerns itself with the inode list. This section lists error conditions resulting
from checking inode types, setting up the zero-link-count table, examining inode block
numbers for bad or duplicate blocks, checking inode size, and checking inode format. All
errors in this phase except **INCORRECT BLOCK COUNT** are fatal if the file system is being
preen'ed,

CG *C*: BAD MAGIC NUMBER The magic number of cylinder group *C* is wrong. This usually
indicates that the cylinder group maps have been destroyed. When running manually the
cylinder group is marked as needing to be reconstructed.

UNKNOWN FILE TYPE I=*I* (CLEAR) The mode word of the inode *I* indicates that the
inode is not a special block inode, special character inode, socket inode, regular inode, symbolic
link, or directory inode.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents. This will always invoke the UNALLOCATED
     error condition in Phase 2 for each directory entry pointing to this inode.

NO   ignore this error condition.

LINK COUNT TABLE OVERFLOW (CONTINUE)
An internal table for *fsck* containing allocated inodes with a link count of zero has no more
room. Recompile *fsck* with a larger value of MAXLNCNT.

Possible responses to the CONTINUE prompt are:

YES continue with the program. This error condition will not allow a complete check of the
     file system. A second run of *fsck* should be made to re-check this file system. If another
     allocated inode with a zero link count is found, this error condition is repeated.

NO   terminate the program.

B BAD I=*I*
Inode *I* contains block number *B* with a number lower than the number of the first data block
in the file system or greater than the number of the last block in the file system. This error
condition may invoke the **EXCESSIVE BAD BLKS** error condition in Phase 1 if inode *I* has
too many block numbers outside the file system range. This error condition will always invoke
the **BAD/DUP** error condition in Phase 2 and Phase 4.

EXCESSIVE BAD BLKS I = I (CONTINUE)
There is more than a tolerable number (usually 10) of blocks with a number lower than the number of the first data block in the file system or greater than the number of last block in the file system associated with inode *I*.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file *system*.

NO   terminate the program.

B DUP I = I
Inode *I* contains block number *B* which is already claimed by another inode. This error condition may invoke the **EXCESSIVE DUP BLKS** error condition in Phase 1 if inode *I* has too many block numbers claimed by other inodes. This error condition will always invoke Phase 1b and the **BAD/DUP** error condition in Phase 2 and Phase 4.

EXCESSIVE DUP BLKS I = I (CONTINUE)
There is more than a tolerable number (usually 10) of blocks claimed by other inodes.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the blocks in this inode and continue checking with the next inode in the file system. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system.

NO   terminate the program.

DUP TABLE OVERFLOW (CONTINUE)
An internal table in *fsck* containing duplicate block numbers has no more room. Recompile *fsck* with a larger value of DUPTBLSIZE.

Possible responses to the CONTINUE prompt are:

YES continue with the program. This error condition will not allow a complete check of the file system. A second run of *fsck* should be made to re-check this file system. If another duplicate block is found, this error condition will repeat.

NO   terminate the program.

PARTIALLY ALLOCATED INODE I = I (CLEAR)
Inode *I* is neither allocated nor unallocated.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO   ignore this error condition.

INCORRECT BLOCK COUNT I = I (*X* should be *Y*) (CORRECT)
The block count for inode *I* is *X* blocks, but should be *Y* blocks. When preen'ing the count is corrected.

Possible responses to the CORRECT prompt are:

YES replace the block count of inode *I* with *Y*.

NO   ignore this error condition.

4.4. Phase 1B: Rescan for More Dups

When a duplicate block is found in the file system, the file system is rescanned to find the inode which previously claimed that block. This section lists the error condition when the duplicate block is found.

B DUP I=*I*

Inode *I* contains block number *B* that is already claimed by another inode. This error condition will always invoke the BAD/DUP error condition in Phase 2. You can determine which inodes have overlapping blocks by examining this error condition and the DUP error condition in Phase 1.

4.5. Phase 2 — Check Pathnames

This phase concerns itself with removing directory entries pointing to error conditioned inodes from Phase 1 and Phase 1b. This section lists error conditions resulting from root inode mode and status, directory inode pointers in range, and directory entries pointing to bad inodes. All errors in this phase are fatal if the file system is being preen'ed.

ROOT INODE UNALLOCATED. TERMINATING.

The root inode (usually inode number 2) has no allocate mode bits. This should never happen. The program will terminate.

NAME TOO LONG *F*

An excessively long path name has been found. This is usually indicative of loops in the file system name space. This can occur if the super user has made circular links to directories. The offending links must be removed (by a guru).

ROOT INODE NOT DIRECTORY (FIX)

The root inode (usually inode number 2) is not directory inode type.

Possible responses to the FIX prompt are:

YES replace the root inode's type to be a directory. If the root inode's data blocks are not directory blocks, a VERY large number of error conditions will be produced.

NO terminate the program.

DUPS/BAD IN ROOT INODE (CONTINUE)

Phase 1 or Phase 1b have found duplicate blocks or bad blocks in the root inode (usually inode number 2) for the file system.

Possible responses to the CONTINUE prompt are:

YES ignore the DUPS/BAD error condition in the root inode and attempt to continue to run the file system check. If the root inode is not correct, then this may result in a large number of other error conditions.

NO terminate the program.

I OUT OF RANGE I=*I* NAME=*F* (REMOVE)

A directory entry *F* has an inode number *I* which is greater than the end of the inode list.

Possible responses to the REMOVE prompt are:

YES the directory entry *F* is removed.

NO ignore this error condition.

UNALLOCATED I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (REMOVE)
A directory entry F has a directory inode I without allocate mode bits. The owner O, mode M, size S, modify time T, and directory name F are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry F is removed.

NO   ignore this error condition.

UNALLOCATED I=I OWNER=O MODE=M SIZE=S MTIME=T FILE=F (REMOVE)
A directory entry F has an inode I without allocate mode bits. The owner O, mode M, size S, modify time T, and file name F are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry F is removed.

NO   ignore this error condition.

DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (REMOVE)
Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory entry F, directory inode I. The owner O, mode M, size S, modify time T, and directory name F are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry F is removed.

NO   ignore this error condition.

DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T FILE=F (REMOVE)
Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory entry F, inode I. The owner O, mode M, size S, modify time T, and file name F are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry F is removed.

NO   ignore this error condition.

ZERO LENGTH DIRECTORY I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (REMOVE)
A directory entry F has a size S that is zero. The owner O, mode M, size S, modify time T, and directory name F are printed.

Possible responses to the REMOVE prompt are:

YES the directory entry F is removed; this will always invoke the BAD/DUP error condition in Phase 4.

NO   ignore this error condition.

DIRECTORY TOO SHORT I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (FIX)
A directory F has been found whose size S is less than the minimum size directory. The owner O, mode M, size S, modify time T, and directory name F are printed.

Possible responses to the FIX prompt are:

YES increase the size of the directory to the minimum directory size.

NO   ignore this directory.

**DIRECTORY CORRUPTED I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F*
(SALVAGE)**
A directory with an inconsistent internal state has been found.

Possible responses to the FIX prompt are:

YES   throw away all entries up to the next 512-byte boundary. This rather drastic action can
         throw away up to 42 entries, and should be taken only after other recovery efforts have
         failed.

NO   Skip up to the next 512-byte boundary and resume reading, but do not modify the direc-
         tory.

**BAD INODE NUMBER FOR '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T*
DIR=*F* (FIX)**
A directory *I* has been found whose inode number for '.' does does not equal *I*.

Possible responses to the FIX prompt are:

YES   change the inode number for '.' to be equal to *I*.

NO   leave the inode number for '.' unchanged.

MISSING '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found whose first entry is unallocated.

Possible responses to the FIX prompt are:

YES   make an entry for '.' with inode number equal to *I*.

NO   leave the directory unchanged.

**MISSING '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F*
CANNOT FIX, FIRST ENTRY IN DIRECTORY CONTAINS *F***
A directory *I* has been found whose first entry is *F*. *Fsck* cannot resolve this problem. The file
system should be mounted and the offending entry *F* moved elsewhere. The file system should
then be unmounted and *fsck* should be run again.

**MISSING '.' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F*
CANNOT FIX, INSUFFICIENT SPACE TO ADD '.'**
A directory *I* has been found whose first entry is not '.'. *Fsck* cannot resolve this problem as it
should never happen. See a guru.

EXTRA '.' ENTRY I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* DIR=*F* (FIX)
A directory *I* has been found that has more than one entry for '.'.

Possible responses to the FIX prompt are:

YES   remove the extra entry for '.'.

NO   leave the directory unchanged.

**BAD INODE NUMBER FOR '..' I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T*
DIR=*F* (FIX)**
A directory *I* has been found whose inode number for '..' does does not equal the parent of *I*.

Possible responses to the FIX prompt are:

YES   change the inode number for '..' to be equal to the parent of *I*.

NO   leave the inode number for '..' unchanged.

MISSING '..' I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (FIX)
A directory I has been found whose second entry is unallocated.

Possible responses to the FIX prompt are:

YES make an entry for '..' with inode number equal to the parent of I.

NO   leave the directory unchanged.

MISSING '..' I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F
CANNOT FIX, SECOND ENTRY IN DIRECTORY CONTAINS F
A directory I has been found whose second entry is F. Fsck cannot resolve this problem. The file system should be mounted and the offending entry F moved elsewhere. The file system should then be unmounted and *fsck* should be run again.

MISSING '..' I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F
CANNOT FIX, INSUFFICIENT SPACE TO ADD '..'
A directory I has been found whose second entry is not '..'. *Fsck* cannot resolve this problem as it should never happen. See a guru.

EXTRA '..' ENTRY I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (FIX)
A directory I has been found that has more than one entry for '..'.

Possible responses to the FIX prompt are:

YES remove the extra entry for '..'.

NO   leave the directory unchanged.

4.6. Phase 3 — Check Connectivity

This phase concerns itself with the directory connectivity seen in Phase 2. This section lists error conditions resulting from unreferenced directories, and missing or full *lost+found* directories.

UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT)
The directory inode I was not connected to a directory entry when the file system was traversed. The owner O, mode M, size S, and modify time T of directory inode I are printed. When preen'ing, the directory is reconnected if its size is non-zero, otherwise it is cleared.

Possible responses to the RECONNECT prompt are:

YES reconnect directory inode I to the file system in the directory for lost files (usually *lost+found*). This may invoke the *lost+found* error condition in Phase 3 if there are problems connecting directory inode I to *lost+found*. This may also invoke the CONNECTED error condition in Phase 3 if the link was successful.

NO   ignore this error condition. This will always invoke the UNREF error condition in Phase 4.

SORRY. NO lost+found DIRECTORY
There is no *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a directory in *lost+found*. This will always invoke the UNREF error condition in Phase 4. Check access modes of *lost+found*. See *fsck* (8) manual entry for further detail. This error is fatal if the file system is being preen'ed.

SORRY. NO SPACE IN lost+found DIRECTORY
There is no space to add another entry to the *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a directory in *lost+found*. This will always invoke the UNREF error condition in Phase 4. Clean out unnecessary entries in *lost+found* or make

lost+found larger. See *fsck* (8) manual entry for further detail. This error is fatal if the file system is being preen'ed.

DIR I=*I1* CONNECTED. PARENT WAS I=*I2*

This is an advisory message indicating a directory inode *I1* was successfully connected to the *lost+found* directory. The parent inode *I2* of the directory inode *I1* is replaced by the inode number of the *lost+found* directory.

4.7. Phase 4 — Check Reference Counts

This phase concerns itself with the link count information seen in Phase 2 and Phase 3. This section lists error conditions resulting from unreferenced files, missing or full *lost+found* directory, incorrect link counts for files, directories, symbolic links, or special files, unreferenced files, symbolic links, and directories, bad and duplicate blocks in files, symbolic links, and directories, and incorrect total free-inode counts. All errors in this phase are correctable if the file system is being preen'ed except running out of space in the lost+found directory.

UNREF FILE I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* (RECONNECT)

Inode *I* was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. When preen'ing the file is cleared if either its size or its link count is zero, otherwise it is reconnected.

Possible responses to the RECONNECT prompt are:

YES reconnect inode *I* to the file system in the directory for lost files (usually *lost+found*). This may invoke the *lost+found* error condition in Phase 4 if there are problems connecting inode *I* to *lost+found*.

NO   ignore this error condition. This will always invoke the CLEAR error condition in Phase 4.

(CLEAR)

The inode mentioned in the immediately previous error condition can not be reconnected. This cannot occur if the file system is being preen'ed, since lack of space to reconnect files is a fatal error.

Possible responses to the CLEAR prompt are:

YES de-allocate the inode mentioned in the immediately previous error condition by zeroing its contents.

NO   ignore this error condition.

SORRY. NO lost+found DIRECTORY

There is no *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a file in *lost+found*. This will always invoke the CLEAR error condition in Phase 4. Check access modes of *lost+found*. This error is fatal if the file system is being preen'ed.

SORRY. NO SPACE IN lost+found DIRECTORY

There is no space to add another entry to the *lost+found* directory in the root directory of the file system; *fsck* ignores the request to link a file in *lost+found*. This will always invoke the CLEAR error condition in Phase 4. Check size and contents of *lost+found*. This error is fatal if the file system is being preen'ed.

LINK COUNT FILE I=*I* OWNER=*O* MODE=*M* SIZE=*S* MTIME=*T* COUNT=*X* SHOULD BE *Y* (ADJUST)

The link count for inode *I* which is a file, is *X* but should be *Y*. The owner *O*, mode *M*, size *S*,

and modify time T are printed. When preen'ing the link count is adjusted.

Possible responses to the ADJUST prompt are:

YES replace the link count of file inode I with Y.

NO   ignore this error condition.

LINK COUNT DIR I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)
The link count for inode I which is a directory, is X but should be Y. The owner O, mode M, size S, and modify time T of directory inode I are printed. When preen'ing the link count is adjusted.

Possible responses to the ADJUST prompt are:

YES replace the link count of directory inode I with Y.

NO   ignore this error condition.

LINK COUNT F I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)
The link count for F inode I is X but should be Y. The name F, owner O, mode M, size S, and modify time T are printed. When preen'ing the link count is adjusted.

Possible responses to the ADJUST prompt are:

YES replace the link count of inode I with Y.

NO   ignore this error condition.

UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)
Inode I which is a file, was not connected to a directory entry when the file system was traversed. The owner O, mode M, size S, and modify time T of inode I are printed. When preen'ing, this is a file that was not connected because its size or link count was zero, hence it is cleared.

Possible responses to the CLEAR prompt are:

YES de-allocate inode I by zeroing its contents.

NO   ignore this error condition.

UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)
Inode I which is a directory, was not connected to a directory entry when the file system was traversed. The owner O, mode M, size S, and modify time T of inode I are printed. When preen'ing, this is a directory that was not connected because its size or link count was zero, hence it is cleared.

Possible responses to the CLEAR prompt are:

YES de-allocate inode I by zeroing its contents.

NO   ignore this error condition.

BAD/DUP FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)
Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with file inode I. The owner O, mode M, size S, and modify time T of inode I are printed. This error cannot arise when the file system is being preen'ed, as it would have caused a fatal error earlier.

Possible responses to the CLEAR prompt are:

YES de-allocate inode I by zeroing its contents.

NO   ignore this error condition.

BAD/DUP DIR I = *I* **OWNER =** *O* **MODE =** *M* **SIZE =** *S* **MTIME =** *T* **(CLEAR)**
Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory inode
I. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. This error cannot
arise when the file system is being preen'ed, as it would have caused a fatal error earlier.

Possible responses to the CLEAR prompt are:

YES de-allocate inode *I* by zeroing its contents.

NO   ignore this error condition.

FREE INODE COUNT WRONG IN SUPERBLK (FIX)
The actual count of the free inodes does not match the count in the super-block of the file sys-
tem. When preen'ing, the count is fixed.

Possible responses to the FIX prompt are:

YES replace the count in the super-block by the actual count.

NO   ignore this error condition.

4.8. Phase 5 - Check Cyl groups

This phase concerns itself with the free-block maps. This section lists error conditions
resulting from allocated blocks in the free-block maps, free blocks missing from free-block
maps, and the total free-block count incorrect.

CG *C*: **BAD MAGIC NUMBER**
The magic number of cylinder group *C* is wrong. This usually indicates that the cylinder group
maps have been destroyed. When running manually the cylinder group is marked as needing to
be reconstructed. This error is fatal if the file system is being preen'ed.

EXCESSIVE BAD BLKS IN BIT MAPS (CONTINUE)
An inode contains more than a tolerable number (usually 10) of blocks claimed by other inodes
or that are out of the legal range for the file system. This error is fatal if the file system is
being preen'ed.

Possible responses to the CONTINUE prompt are:

YES ignore the rest of the free-block maps and continue the execution of *fsck*.

NO   terminate the program.

SUMMARY INFORMATION *T* **BAD**
where *T* is one or more of:
(INODE FREE)
(BLOCK OFFSETS)
(FRAG SUMMARIES)
(SUPER BLOCK SUMMARIES)
The indicated summary information was found to be incorrect. This error condition will always
invoke the **BAD CYLINDER GROUPS** condition in Phase 6. When preen'ing, the summary
information is recomputed.

X **BLK(S) MISSING**
X blocks unused by the file system were not found in the free-block maps. This error condition
will always invoke the **BAD CYLINDER GROUPS** condition in Phase 6. When preen'ing, the
block maps are rebuilt.

BAD CYLINDER GROUPS (SALVAGE)
Phase 5 has found bad blocks in the free-block maps, duplicate blocks in the free-block maps,

or blocks missing from the file system. When preen'ing, the cylinder groups are reconstructed.

Possible responses to the SALVAGE prompt are:

YES replace the actual free-block maps with a new free-block maps.

NO ignore this error condition.

FREE BLK COUNT WRONG IN SUPERBLOCK (FIX)
The actual count of free blocks does not match the count in the super-block of the file system. When preen'ing, the counts are fixed.

Possible responses to the FIX prompt are:

YES replace the count in the super-block by the actual count.

NO ignore this error condition.

4.9. Phase 6 - Salvage Cylinder Groups

This phase concerns itself with the free-block maps reconstruction. No error messages are produced.

4.10. Cleanup

Once a file system has been checked, a few cleanup functions are performed. This section lists advisory messages about the file system and modify status of the file system.

V files, W used, X free (Y frags, Z blocks)
This is an advisory message indicating that the file system checked contained V files using W fragment sized blocks leaving X fragment sized blocks free in the file system. The numbers in parenthesis breaks the free count down into Y free fragments and Z free full sized blocks.

***** REBOOT UNIX *****
This is an advisory message indicating that the root file system has been modified by *fsck*. If UNIX is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX keeps. When preen'ing, *fsck* will exit with a code of 4. The auto-reboot script interprets an exit code of 4 by issuing a reboot system call.

***** FILE SYSTEM WAS MODIFIED *****
This is an advisory message indicating that the current file system was modified by *fsck*. If this file system *is mounted* or is the current root file system, *fsck* should be halted and UNIX rebooted. If UNIX is not rebooted immediately, the work done by *fsck* may be undone by the in-core copies of tables UNIX keeps.

SENDMAIL

INSTALLATION AND OPERATION GUIDE

Eric Allman
Britton-Lee, Inc.

Version 4.2

TABLE OF CONTENTS

SENDMAIL

INSTALLATION AND OPERATION GUIDE

Eric Allman
Britton-Lee, Inc.

Version 4.2

Sendmail implements a general purpose internetwork mail routing facility under the UNIX* operating system. It is not tied to any one transport protocol — its function may be likened to a crossbar switch, relaying messages from one domain into another. In the process, it can do a limited amount of message header editing to put the message into a format that is appropriate for the receiving domain. All of this is done under the control of a configuration file.

Due to the requirements of flexibility for *sendmail*, the configuration file can seem somewhat unapproachable. However, there are only a few basic configurations for most sites, for which standard configuration files have been supplied. Most other configurations can be built by adjusting an existing configuration files incrementally.

Although *sendmail* is intended to run without the need for monitoring, it has a number of features that may be used to monitor or adjust the operation under unusual circumstances. These features are described.

Section one describes how to do a basic *sendmail* installation. Section two explains the day-to-day information you should know to maintain your mail system. If you have a relatively normal site, these two sections should contain sufficient information for you to install *sendmail* and keep it happy. Section three describes some parameters that may be safely tweaked. Section four has information regarding the command line arguments. Section five contains the nitty-gritty information about the configuration file. This section is for masochists and people who must write their own configuration file. The appendixes give a brief but detailed explanation of a number of features not described in the rest of the paper.

The references in this paper are actually found in the companion paper *Sendmail — An Internetwork Mail Router*. This other paper should be read before this manual to gain a basic understanding of how the pieces fit together.

1. BASIC INSTALLATION

There are two basic steps to installing sendmail. The hard part is to build the configuration table. This is a file that sendmail reads when it starts up that describes the mailers it knows about, how to parse addresses, how to rewrite the message header, and the settings of various options. Although .the configuration table is quite complex, a configuration can usually be built by adjusting an existing off-the-shelf configuration. The second part is actually doing the installation, i.e., creating the necessary files, etc.

The remainder of this section will describe the installation of sendmail assuming you can use one of the existing configurations and that the standard installation parameters are acceptable. All pathnames and examples are given from the root of the *sendmail* subtree.

---

*UNIX is a trademark of Bell Laboratories.

1.1. Off-The-Shelf Configurations

The configuration files are all in the subdirectory *cf* of the sendmail directory. The ones used at Berkeley are in *m4* (1) format; files with names ending ".m4" are *m4* include files, while files with names ending ".mc" are the master files. Files with names ending ".cf" are the *m4* processed versions of the corresponding ".mc" file.

Two off the shelf configuration files are supplied to handle the basic cases: *cf/arpaproto.cf* for Arpanet (TCP) sites and *cf/uucpproto.cf* for UUCP sites. These are *not* in *m4* format. The file you need should be copied to a file with the same name as your system, e.g.,

    cp uucpproto.cf ucsfcgl.cf

This file is now ready for installation as */usr/lib/sendmail.cf.*

1.2. Installation Using the Makefile

A makefile exists in the root of the *sendmail* directory that will do all of these steps for a 4.2BSD system. It may have to be slightly tailored for use on other systems.

Before using this makefile, you should already have created your configuration file and left it in the file "cf/*system*.cf" where *system* is the name of your system (i.e., what is returned by *hostname* (1)). If you do not have *hostname* you can use the declaration "HOST=*system*" on the *make* (1) command line. You should also examine the file *md/config.m4* and change the *m4* macros there to reflect any libraries and compilation flags you may need.

The basic installation procedure is to type:

    make
    make install

in the root directory of the *sendmail* distribution. This will make all binaries and install them in the standard places. The second *make* command must be executed as the superuser (root).

1.3. Installation by Hand

Along with building a configuration file, you will have to install the *sendmail* startup into your UNIX system. If you are doing this installation in conjunction with a regular Berkeley UNIX install, these steps will already be complete. Many of these steps will have to be executed as the superuser (root).

1.3.1. lib/libsys.a

The library in lib/libsys.a contains some routines that should in some sense be part of the system library. These are the system logging routines and the new directory access routines (if required). If you are not running the new 4.2BSD directory code and do not have the compatibility routines installed in your system library, you should execute the commands:

    cd lib
    make ndir

This will compile and install the 4.2 compatibility routines in the library. You should then type:

    cd lib   # if required
    make

This will recompile and fill the library.

1.3.2. /usr/lib/sendmail

The binary for sendmail is located in /usr/lib. There is a version available in the source directory that is probably inadequate for your system. You should plan on recompiling and installing the entire system:

```
cd src
rm −f *.o
make
cp sendmail /usr/lib
```

1.3.3. /usr/lib/sendmail.cf

The configuration file that you created earlier should be installed in /usr/lib/sendmail.cf:

```
cp cf/system.cf /usr/lib/sendmail.cf
```

1.3.4. /usr/ucb/newaliases

If you are running delivermail, it is critical that the *newaliases* command be replaced. This can just be a link to *sendmail:*

```
rm −f /usr/ucb/newaliases
ln /usr/lib/sendmail /usr/ucb/newaliases
```

1.3.5. /usr/lib/sendmail.cf

The configuration file must be installed in /usr/lib. This is described above.

1.3.6. /usr/spool/mqueue

The directory */usr/spool/mqueue* should be created to hold the mail queue. This directory should be mode 777 unless *sendmail* is run setuid, when *mqueue* should be owned by the sendmail owner and mode 755.

1.3.7. /usr/lib/aliases*

The system aliases are held in three files. The file "/usr/lib/aliases" is the master copy. A sample is given in "lib/aliases" which includes some aliases which *must* be defined:

```
cp lib/aliases /usr/lib/aliases
```

You should extend this file with any aliases that are apropos to your system.

Normally *sendmail* looks at a version of these files maintained by the *dbm*(3) routines. These are stored in "/usr/lib/aliases.dir" and "/usr/lib/aliases.pag." These can initially be created as empty files, but they will have to be initialized promptly. These should be mode 666 if you are running a reasonably relaxed system:

```
cp /dev/null /usr/lib/aliases.dir
cp /dev/null /usr/lib/aliases.pag
chmod 666 /usr/lib/aliases.*
newaliases
```

1.3.8. /usr/lib/sendmail.fc

If you intend to install the frozen version of the configuration file (for quick startup) you should create the file /usr/lib/sendmail.fc and initialize it. This step may be safely skipped.

```
cp /dev/null /usr/lib/sendmail.fc
/usr/lib/sendmail —bz
```

1.3.9. /etc/rc

It will be necessary to start up the sendmail daemon when your system reboots. This daemon performs two functions: it listens on the SMTP socket for connections (to receive mail from a remote system) and it processes the queue periodically to insure that mail gets delivered when hosts come up.

Add the following lines to "/etc/rc" (or "/etc/rc.local" as appropriate) in the area where it is starting up the daemons:

```
if [ —f /usr/lib/sendmail ]; then
        (cd /usr/spool/mqueue; rm —f [lnx]f*)
        /usr/lib/sendmail —bd —q30m &
        echo —n ' sendmail' >/dev/console
fi
```

The "cd" and "rm" commands insure that all lock files have been removed; extraneous lock files may be left around if the system goes down in the middle of processing a message. The line that actually invokes *sendmail* has two flags: "—bd" causes it to listen on the SMTP port, and "—q30m" causes it to run the queue every half hour.

If you are not running a version of UNIX that supports Berkeley TCP/IP, do not include the —bd flag.

1.3.10. /usr/lib/sendmail.hf

This is the help file used by the SMTP HELP command. It should be copied from "lib/sendmail.hf":

```
cp lib/sendmail.hf /usr/lib
```

1.3.11. /usr/lib/sendmail.st

If you wish to collect statistics about your mail traffic, you should create the file "/usr/lib/sendmail.st":

```
cp /dev/null /usr/lib/sendmail.st
chmod 666 /usr/lib/sendmail.st
```

This file does not grow. It is printed with the program "aux/mailstats."

1.3.12. /etc/syslog

You may want to run the *syslog* program (to collect log information about sendmail). This program normally resides in */etc/syslog*, with support files */etc/syslog.conf* and */etc/syslog.pid*. The program is located in the *aux* subdirectory of the *sendmail* distribution. The file */etc/syslog.conf* describes the file(s) that sendmail will log in. For a complete description of syslog, see the manual page for *syslog*(8) (located in *sendmail/doc* on the distribution).

1.3.13. /usr/ucb/newaliases

If *sendmail* is invoked as "newaliases," it will simulate the —bi flag (i.e., will rebuild the alias database; see below). This should be a link to /usr/lib/sendmail.

1.3.14. /usr/ucb/mailq

If *sendmail* is invoked as "mailq," it will simulate the —bp flag (i.e., *sendmail* will print the contents of the mail queue; see below). This should be a link to /usr/lib/sendmail.

2. NORMAL OPERATIONS

2.1. Quick Configuration Startup

A fast version of the configuration file may be set up by using the **−bz** flag:

/usr/lib/sendmail −bz

This creates the file */usr/lib/sendmail.fc* ("frozen configuration"). This file is an image of *sendmail*'s data space after reading in the configuration file. If this file exists, it is used instead of */usr/lib/sendmail.cf sendmail.fc* must be rebuilt manually every time *sendmail.cf* is changed.

The frozen configuration file will be ignored if a **−C** flag is specified or if sendmail detects that it is out of date. However, the heuristics are not strong so this should not be trusted.

2.2. The System Log

The system log is supported by the *syslog* (8) program.

2.2.1. Format

Each line in the system log consists of a timestamp, the name of the machine that generated it (for logging from several machines over the ethernet), the word "sendmail:", and a message.

2.2.2. Levels

If you have *syslog* (8) or an equivalent installed, you will be able to do logging. There is a large amount of information that can be logged. The log is arranged as a succession of levels. At the lowest level only extremely strange situations are logged. At the highest level, even the most mundane and uninteresting events are recorded for posterity. As a convention, log levels under ten are considered "useful;" log levels above ten are usually for debugging purposes.

A complete description of the log levels is given in section 4.3.

2.3. The Mail Queue

The mail queue should be processed transparently. However, you may find that manual intervention is sometimes necessary. For example, if a major host is down for a period of time the queue may become clogged. Although sendmail ought to recover gracefully when the host comes up, you may find performance unacceptably bad in the meantime.

2.3.1. Printing the queue

The contents of the queue can be printed using the *mailq* command (or by specifying the **−bp** flag to sendmail):

mailq

This will produce a listing of the queue id's, the size of the message, the date the message entered the queue, and the sender and recipients.

2.3.2. Format of queue files

All queue files have the form *xfAA99999* where *AA99999* is the *id* for this file and the *x* is a type. The types are:

d   The data file. The message body (excluding the header) is kept in this file.

l   The lock file. If this file exists, the job is currently being processed, and a queue run will not process the file. For that reason, an extraneous lf file can

cause a job to apparently disappear (it will not even time out!).

n   This file is created when an id is being created. It is a separate file to insure that no mail can ever be destroyed due to a race condition. It should exist for no more than a few milliseconds at any given time.

q   The queue control file. This file contains the information necessary to process the job.

t   A temporary file. These are an image of the qf file when it is being rebuilt. It should be renamed to a qf file very quickly.

x   A transcript file, existing during the life of a session showing everything that happens during that session.

The qf file is structured as a series of lines each beginning with a code letter. The lines are as follows:

D   The name of the data file. There may only be one of these lines.

H   A header definition. There may be any number of these lines. The order is important: they represent the order in the final message. These use the same syntax as header definitions in the configuration file.

R   A recipient address. This will normally be completely aliased, but is actually realiased when the job is processed. There will be one line for each recipient.

S   The sender address. There may only be one of these lines.

T   The job creation time. This is used to compute when to time out the job.

P   The current message priority. This is used to order the queue. Higher numbers mean lower priorities. The priority increases as the message sits in the queue. The initial priority depends on the message class and the size of the message.

M   A message. This line is printed by the *mailq* command, and is generally used to store status information. It can contain any text.

As an example, the following is a queue file sent to "mckusick@calder" and "wnj":

```
DdfA13557
Seric
T404261372
P132
Rmckusick@calder
Rwnj
H?D?date: 23-Oct-82 15:49:32-PDT (Sat)
H?F?from: eric (Eric Allman)
H?x?full-name: Eric Allman
Hsubject: this is an example message
Hmessage-id: <8209232249.13557@UCBARPA.BERKELEY.ARPA>
Hreceived: by UCBARPA.BERKELEY.ARPA (3.227 [10/22/82])
        id A13557; 23-Oct-82 15:49:32-PDT (Sat)
Hphone: (415) 548-3211
HTo: mckusick@calder, wnj
```

This shows the name of the data file, the person who sent the message, the submission time (in seconds since January 1, 1970), the message priority, the message class, the recipients, and the headers for the message.

2.3.3. Forcing the queue

Sendmail should run the queue automatically at intervals. The algorithm is to read and sort the queue, and then to attempt to process all jobs in order. When it attempts to run the job, *sendmail* first checks to see if the job is locked. If so, it

ignores the job.

There is no attempt to insure that only one queue processor exists at any time, since there is no guarantee that a job cannot take forever to process. Due to the locking algorithm, it is impossible for one job to freeze the queue. However, an uncooperative recipient host or a program recipient that never returns can accumulate many processes in your system. Unfortunately, there is no way to resolve this without violating the protocol.

In some cases, you may find that a major host going down for a couple of days may create a prohibitively large queue. This will result in *sendmail* spending an inordinate amount of time sorting the queue. This situation can be fixed by moving the queue to a temporary place and creating a new queue. The old queue can be run later when the offending host returns to service.

To do this, it is acceptable to move the entire queue directory:

cd /usr/spool
mv mqueue omqueue; mkdir mqueue; chmod 777 mqueue

You should then kill the existing daemon (since it will still be processing in the old queue directory) and create a new daemon.

To run the old mail queue, run the following command:

/usr/lib/sendmail −oQ/usr/spool/omqueue −q

The −oQ flag specifies an alternate queue directory and the −q flag says to just run every job in the queue. If you have a tendency toward voyeurism, you can use the −v flag to watch what is going on.

When the queue is finally emptied, you can remove the directory:

rmdir /usr/spool/omqueue

2.4. The Alias Database

The alias database exists in two forms. One is a text form, maintained in the file */usr/lib/aliases*. The aliases are of the form

name: name1, name2, ...

Only local names may be aliased; e.g.,

eric@mit-xx: eric@berkeley

will not have the desired effect. Aliases may be continued by starting any continuation lines with a space or a tab. Blank lines and lines beginning with a sharp sign ("#") are comments.

The second form is processed by the *dbm*(3) library. This form is in the files */usr/lib/aliases.dir* and */usr/lib/aliases.pag*. This is the form that *sendmail* actually uses to resolve aliases. This technique is used to improve performance.

2.4.1. Rebuilding the alias database

The DBM version of the database may be rebuilt explicitly by executing the command

newaliases

This is equivalent to giving *sendmail* the −bi flag:

/usr/lib/sendmail −bi

If the "D" option is specified in the configuration, *sendmail* will rebuild the alias database automatically if possible when it is out of date. The conditions under which it will do this are:

(1)   The DBM version of the database is mode 666.   -or-

(2)   *Sendmail* is running setuid to root.

Auto-rebuild can be dangerous on heavily loaded machines with large alias files; if it might take more than five minutes to rebuild the database, there is a chance that several processes will start the rebuild process simultaneously.

2.4.2. Potential problems

There are a number of problems that can occur with the alias database. They all result from a *sendmail* process accessing the DBM version while it is only partially built. This can happen under two circumstances: One process accesses the database while another process is rebuilding it, or the process rebuilding the database dies (due to being killed or a system crash) before completing the rebuild.

Sendmail has two techniques to try to relieve these problems. First, it ignores interrupts while rebuilding the database; this avoids the problem of someone aborting the process leaving a partially rebuilt database. Second, at the end of the rebuild it adds an alias of the form

@: @

(which is not normally legal). Before sendmail will access the database, it checks to insure that this entry exists[1]. It will wait up to five minutes for this entry to appear, at which point it will force a rebuild itself[2].

2.4.3. List owners

If an error occurs on sending to a certain address, say "*x*", *sendmail* will look for an alias of the form "owner-*x*" to receive the errors. This is typically useful for a mailing list where the submitter of the list has no control over the maintenance of the list itself; in this case the list maintainer would be the owner of the list. For example:

    unix-wizards: eric@ucbarpa, wnj@monet, nosuchuser,
        sam@matisse
    owner-unix-wizards: eric@ucbarpa

would cause "eric@ucbarpa" to get the error that will occur when someone sends to unix-wizards due to the inclusion of "nosuchuser" on the list.

2.5. Per-User Forwarding (.forward Files)

As an alternative to the alias database, any user may put a file with the name ".forward" in his or her home directory. If this file exists, *sendmail* redirects mail for that user to the list of addresses listed in the .forward file. For example, if the home directory for user "mckusick" has a .forward file with contents:

    mckusick@ernie
    kirk@calder

then any mail arriving for "mckusick" will be redirected to the specified accounts.

2.6. Special Header Lines

Several header lines have special interpretations defined by the configuration file. Others have interpretations built into *sendmail* that cannot be changed without changing the code. These builtins are described here.

---

[1]The "a" option is required in the configuration for this action to occur. This should normally be specified unless you are running *delivermail* in parallel with *sendmail.*

[2]Note: the "D" option must be specified in the configuration file for this operation to occur.

2.6.1. Return-Receipt-To:

If this header is sent, a message will be sent to any specified addresses when the final delivery is complete. if the mailer has the l flag (local delivery) set in the mailer descriptor.

2.6.2. Errors-To:

If errors occur anywhere during processing, this header will cause error messages to go to the listed addresses rather than to the sender. This is intended for mailing lists.

2.6.3. Apparently-To:

If a message comes in with no recipients listed in the message (in a To:, Cc:, or Bcc: line) then *sendmail* will add an "Apparently-To:" header line for any recipients it is aware of. This is not put in as a standard recipient line to warn any recipients that the list is not complete.

At least one recipient line is required under RFC 822.

3. ARGUMENTS

The complete list of arguments to *sendmail* is described in detail in Appendix A. Some important arguments are described here.

3.1. Queue Interval

The amount of time between forking a process to run through the queue is defined by the −q flag. If you run in mode f or a this can be relatively large, since it will only be relevant when a host that was down comes back up. If you run in q mode it should be relatively short, since it defines the maximum amount of time that a message may sit in the queue.

3.2. Daemon Mode

If you allow incoming mail over an IPC connection, you should have a daemon running. This should be set by your */etc/rc* file using the −bd flag. The −bd flag and the −q flag may be combined in one call:

    /usr/lib/sendmail −bd −q30m

3.3. Forcing the Queue

In some cases you may find that the queue has gotten clogged for some reason. You can force a queue run using the −q flag (with no value). It is entertaining to use the −v flag (verbose) when this is done to watch what happens:

    /usr/lib/sendmail −q −v

3.4. Debugging

There are a fairly large number of debug flags built into *sendmail*. Each debug flag has a number and a level, where higher levels means to print out more information. The convention is that levels greater than nine are "absurd," i.e., they print out so much information that you wouldn't normally want to see them except for debugging that particular piece of code. Debug flags are set using the −d option; the syntax is:

```
debug-flag:   −d debug-list
debug-list:   debug-option [ , debug-option ]
debug-option: debug-range [ . debug-level ]
debug-range:  integer | integer − integer
debug-level:  integer
```

where spaces are for reading ease only. For example,

```
−d12        Set flag 12 to level 1
−d12.3      Set flag 12 to level 3
−d3-17      Set flags 3 through 17 to level 1
−d3-17.4    Set flags 3 through 17 to level 4
```

For a complete list of the available debug flags you will have to look at the code (they are too dynamic to keep this documentation up to date).

3.5. Trying a Different Configuration File

An alternative configuration file can be specified using the −C flag; for example,

/usr/lib/sendmail −Ctest.cf

uses the configuration file *test.cf* instead of the default */usr/lib/sendmail.cf.* If the −C flag has no value it defaults to *sendmail.cf* in the current directory.

3.6. Changing the Values of Options

Options can be overridden using the −o flag. For example,

/usr/lib/sendmail −oT2m

sets the T (timeout) option to two minutes for this run only.

4. TUNING

There are a number of configuration parameters you may want to change, depending on the requirements of your site. Most of these are set using an option in the configuration file. For example, the line "OT3d" sets option "T" to the value "3d" (three days).

4.1. Timeouts

All time intervals are set using a scaled syntax. For example, "10m" represents ten minutes, whereas "2h30m" represents two and a half hours. The full set of scales is:

```
s    seconds
m    minutes
h    hours
d    days
w    weeks
```

4.1.1. Queue interval

The argument to the −q flag specifies how often a subdaemon will run the queue. This is typically set to between five minutes and one half hour.

4.1.2. Read timeouts

It is possible to time out when reading the standard input or when reading from a remote SMTP server. Technically, this is not acceptable within the published protocols. However, it might be appropriate to set it to something large in certain environments (such as an hour). This will reduce the chance of large numbers of idle daemons piling up on your system. This timeout is set using the r option in the configuration file.

4.1.3. Message timeouts

After sitting in the queue for a few days, a message will time out. This is to insure that at least the sender is aware of the inability to send a message. The timeout is typically set to three days. This timeout is set using the T option in the configuration file.

The time of submission is set in the queue, rather than the amount of time left until timeout. As a result, you can flush messages that have been hanging for a short period by running the queue with a short message timeout. For example,

        /usr/lib/sendmail −oT1d −q

will run the queue and flush anything that is one day old.

4.2. Delivery Mode

There are a number of delivery modes that *sendmail* can operate in, set by the "d" configuration option. These modes specify how quickly mail will be delivered. Legal modes are:

  i   deliver interactively (synchronously)
  b   deliver in background (asynchronously)
  q   queue only (don't deliver)

There are tradeoffs. Mode "i" passes the maximum amount of information to the sender, but is hardly ever necessary. Mode "q" puts the minimum load on your machine, but means that delivery may be delayed for up to the queue interval. Mode "b" is probably a good compromise. However, this mode can cause large numbers of processes if you have a mailer that takes a long time to deliver a message.

4.3. Log Level

The level of logging can be set for sendmail. The default using a standard configuration table is level 9. The levels are as follows:

  0   No logging.

  1   Major problems only.

  2   Message collections and failed deliveries.

  3   Successful deliveries.

  4   Messages being defered (due to a host being down, etc.).

  5   Normal message queueups.

  6   Unusual but benign incidents, e.g., trying to process a locked queue file.

  9   Log internal queue id to external message id mappings. This can be useful for tracing a message as it travels between several hosts.

  12   Several messages that are basically only of interest when debugging.

  16   Verbose information regarding the queue.

4.4. File Modes

There are a number of files that may have a number of modes. The modes depend on what functionality you want and the level of security you require.

4.4.1. To suid or not to suid?

Sendmail can safely be made setuid to root. At the point where it is about to *exec*(2) a mailer, it checks to see if the userid is zero; if so, it resets the userid and groupid to a default (set by the u and g options). (This can be overridden by setting the S flag to the mailer for mailers that are trusted and must be called as root.)

However, this will cause mail processing to be accounted (using *sa* (8)) to root rather than to the user sending the mail.

4.4.2. Temporary file modes

The mode of all temporary files that *sendmail* creates is determined by the "F" option. Reasonable values for this option are 0600 and 0644. If the more permissive mode is selected, it will not be necessary to run *sendmail* as root at all (even when running the queue).

4.4.3. Should my alias database be writable?

At Berkeley we have the alias database (/usr/lib/aliases*) mode 666. There are some dangers inherent in this approach: any user can add him-/her-self to any list, or can "steal" any other user's mail. However, we have found users to be basically trustworthy, and the cost of having a read-only database greater than the expense of finding and eradicating the rare nasty person.

The database that *sendmail* actually used is represented by the two files *aliases.dir* and *aliases.pag* (both in /usr/lib). The mode on these files should match the mode on /usr/lib/aliases. If *aliases* is writable and the DBM files (*aliases.dir* and *aliases.pag*) are not, users will be unable to reflect their desired changes through to the actual database. However, if *aliases* is read-only and the DBM files are writable, a slightly sophisticated user can arrange to steal mail anyway.

If your DBM files are not writable by the world or you do not have auto-rebuild enabled (with the "D" option), then you must be careful to reconstruct the alias database each time you change the text version:

    newaliases

If this step is ignored or forgotten any intended changes will also be ignored or forgotten.

5. THE WHOLE SCOOP ON THE CONFIGURATION FILE

This section describes the configuration file in detail, including hints on how to write one of your own if you have to.

There is one point that should be made clear immediately: the syntax of the configuration file is designed to be reasonably easy to parse, since this is done every time *sendmail* starts up, rather than easy for a human to read or write. On the "future project" list is a configuration-file compiler.

An overview of the configuration file is given first, followed by details of the semantics.

5.1. The Syntax

The configuration file is organized as a series of lines, each of which begins with a single character defining the semantics for the rest of the line. Lines beginning with a space or a tab are continuation lines (although the semantics are not well defined in many places). Blank lines and lines beginning with a sharp symbol ('#') are comments.

5.1.1. R and S — rewriting rules

The core of address parsing are the rewriting rules. These are an ordered production system. *Sendmail* scans through the set of rewriting rules looking for a match on the left hand side (LHS) of the rule. When a rule matches, the address is replaced by the right hand side (RHS) of the rule.

There are several sets of rewriting rules. Some of the rewriting sets are used internally and must have specific semantics. Other rewriting sets do not have

specifically assigned semantics, and may be referenced by the mailer definitions or by other rewriting sets.

The syntax of these two commands are:

S*n*

Sets the current ruleset being collected to *n*. If you begin a ruleset more than once it deletes the old definition.

R *lhs rhs comments*

The fields must be separated by at least one tab character; there may be embedded spaces in the fields. The *lhs* is a pattern that is applied to the input. If it matches, the input is rewritten to the *rhs*. The *comments* are ignored.

5.1.2. D — define macro

Macros are named with a single character. These may be selected from the entire ASCII set, but user-defined macros should be selected from the set of upper case letters only. Lower case letters and special symbols are used internally.

The syntax for macro definitions is:

D*x val*

where *x* is the name of the macro and *val* is the value it should have. Macros can be interpolated in most places using the escape sequence $*x*.

5.1.3. C and F — define classes

Classes of words may be defined to match on the left hand side of rewriting rules. For example a class of all local names for this site might be created so that attempts to send to oneself can be eliminated. These can either be defined directly in the configuration file or read in from another file. Classes may be given names from the set of upper case letters. Lower case letters and special characters are reserved for system use.

The syntax is:

C*c word1 word2...*
F*c file* [*format*]

The first form defines the class *c* to match any of the named words. It is permissible to split them among multiple lines; for example, the two forms:

CHmonet ucbmonet

and

CHmonet
CHucbmonet

are equivalent. The second form reads the elements of the class *c* from the named *file*; the *format* is a *scanf*(3) pattern that should produce a single string.

5.1.4. M — define mailer

Programs and interfaces to mailers are defined in this line. The format is:

M*name,* {*field*=*value*}*

where *name* is the name of the mailer (used internally only) and the "field=name" pairs define attributes of the mailer. Fields are:

Path	The pathname of the mailer
Flags	Special flags for this mailer
Sender	A rewriting set for sender addresses
Recipient	A rewriting set for recipient addresses
Argv	An argument vector to pass to this mailer
Eol	The end-of-line string for this mailer
Maxsize	The maximum message length to this mailer

Only the first character of the field name is checked.

5.1.5. H — define header

The format of the header lines that sendmail inserts into the message are defined by the H line. The syntax of this line is:

H[? *mflags*?] *hname*: *htemplate*

Continuation lines in this spec are reflected directly into the outgoing message. The *htemplate* is macro expanded before insertion into the message. If the *mflags* (surrounded by question marks) are specified, at least one of the specified flags must be stated in the mailer definition for this header to be automatically output. If one of these headers is in the input it is reflected to the output regardless of these flags.

Some headers have special semantics that will be described below.

5.1.6. O — set option

There are a number of "random" options that can be set from a configuration file. Options are represented by single characters. The syntax of this line is:

O*o value*

This sets option *o* to be *value*. Depending on the option, *value* may be a string, an integer, a boolean (with legal values "t", "T", "f", or "F"; the default is TRUE), or a time interval.

5.1.7. T — define trusted users

Trusted users are those users who are permitted to override the sender address using the −f flag. These typically are "root," "uucp," and "network," but on some users it may be convenient to extend this list to include other users, perhaps to support a separate UUCP login for each host. The syntax of this line is:

T*user1 user2...*

There may be more than one of these lines.

5.1.8. P — precedence definitions

Values for the "Precedence:" field may be defined using the P control line. The syntax of this field is:

P*name*=*num*

When the *name* is found in a "Precedence:" field, the message class is set to *num*. Higher numbers mean higher precedence. Numbers less than zero have the special property that error messages will not be returned. The default precedence is zero. For example, our list of precedences is:

Pfirst-class=0
Pspecial-delivery=100
Pjunk=−100

5.2. The Semantics

This section describes the semantics of the configuration file.

5.2.1. Special macros, conditionals

Macros are interpolated using the construct x, where x is the name of the macro to be interpolated. In particular, lower case letters are reserved to have special semantics, used to pass information in or out of sendmail, and some special characters are reserved to provide conditionals, etc.

The following macros *must* be defined to transmit information into *sendmail:*

e   The SMTP entry message
j   The "official" domain name for this site
l   The format of the UNIX from line
n   The name of the daemon (for error messages)
o   The set of "operators" in addresses
q   default format of sender address

The $e macro is printed out when SMTP starts up. The first word must be the $j macro. The $j macro should be in RFC821 format. The $l and $n macros can be considered constants except under terribly unusual circumstances. The $o macro consists of a list of characters which will be considered tokens and which will separate tokens when doing parsing. For example, if "r" were in the $o macro, then the input "address" would be scanned as three tokens: "add," "r," and "ess." Finally, the $q macro specifies how an address should appear in a message when it is defaulted. For example, on our system these definitions are:

```
De$j Sendmail $v ready at $b
DnMAILER-DAEMON
DlFrom $g  $d
Do.:%@!^=/
Dq$g$?x ($x)$.
Dj$H.$D
```

An acceptable alternative for the $q macro is "$?x$x $.<$g>". These correspond to the following two formats:

```
eric@Berkeley (Eric Allman)
Eric Allman <eric@Berkeley>
```

Some macros are defined by *sendmail* for interpolation into argv's for mailers or for other contexts. These macros are:

a   The origination date in Arpanet format
b   The current date in Arpanet format
c   The hop count
d   The date in UNIX (ctime) format
f   The sender (from) address
g   The sender address relative to the recipient
h   The recipient host
i   The queue id
p   Sendmail's pid
r   Protocol used
s   Sender's host name
t   A numeric representation of the current time
u   The recipient user
v   The version number of sendmail
w   The hostname of this site
x   The full name of the sender
y   The id of the sender's tty
z   The home directory of the recipient

There are three types of dates that can be used. The $a and $b macros are in Arpanet format; $a is the time as extracted from the "Date:" line of the message (if there was one), and $b is the current date and time (used for postmarks). If no "Date:" line is found in the incoming message, $a is set to the current time also. The $d macro is equivalent to the $a macro in UNIX (ctime) format.

The $f macro is the id of the sender as originally determined; when mailing to a specific host the $g macro is set to the address of the sender *relative to the recipient*. For example, if I send to "bollard@matisse" from the machine "ucbarpa" the $f macro will be "eric" and the $g macro will be "eric@ucbarpa."

The $x macro is set to the full name of the sender. This can be determined in several ways. It can be passed as flag to *sendmail*. The second choice is the value of the "Full-name:" line in the header if it exists, and the third choice is the comment field of a "From:" line. If all of these fail, and if the message is being originated locally, the full name is looked up in the */etc/passwd* file.

When sending, the $h, $u, and $z macros get set to the host, user, and home directory (if local) of the recipient. The first two are set from the $@ and $: part of the rewriting rules, respectively.

The $p and $t macros are used to create unique strings (e.g., for the "Message-Id:" field). The $i macro is set to the queue id on this host; if put into the timestamp line it can be extremely useful for tracking messages. The $y macro is set to the id of the terminal of the sender (if known); some systems like to put this in the Unix "From" line. The $v macro is set to be the version number of *sendmail*; this is normally put in timestamps and has been proven extremely useful for debugging. The $w macro is set to the name of this host if it can be determined. The $c field is set to the "hop count," i.e., the number of times this message has been processed. This can be determined by the −h flag on the command line or by counting the timestamps in the message.

The $r and $s fields are set to the protocol used to communicate with sendmail and the sending hostname; these are not supported in the current version.

Conditionals can be specified using the syntax:

$?x text1 $| text2 $.

This interpolates *text1* if the macro $x is set, and *text2* otherwise. The "else" ($|) clause may be omitted.

5.2.2. Special classes

The class **$=w** is set to be the set of all names this host is known by. This can be used to delete local hostnames.

5.2.3. The left hand side

The left hand side of rewriting rules contains a pattern. Normal words are simply matched directly. Metasyntax is introduced using a dollar sign. The metasymbols are:

$*   Match zero or more tokens
$+   Match one or more tokens
$−   Match exactly one token
**$=**$x$ Match any token in class x
**$⁻**$x$ Match any token not in class x

If any of these match, they are assigned to the symbol **$**$n$ for replacement on the right hand side, where n is the index in the LHS. For example, if the LHS:

    $−:$+

is applied to the input:

    UCBARPA:eric

the rule will match, and the values passed to the RHS will be:

    $1   UCBARPA
    $2   eric

5.2.4. The right hand side

When the right hand side of a rewriting rule matches, the input is deleted and replaced by the right hand side. Tokens are copied directly from the RHS unless they are begin with a dollar sign. Metasymbols are:

**$**$n$ Substitute indefinite token n from LHS
**$>**$n$ "Call" ruleset n
$# *mailer*   Resolve to *mailer*
$@ *host*   Specify *host*
**$:**$user$ Specify *user*

The **$**$n$ syntax substitutes the corresponding value from a **$+**, **$−**, **$***, **$=**, or **$⁻** match on the LHS. It may be used anywhere.

The **$>**$n$ syntax causes the remainder of the line to be substituted as usual and then passed as the argument to ruleset n. The final value of ruleset n then becomes the substitution for this rule.

The **$#** syntax should *only* be used in ruleset zero. It causes evaluation of the ruleset to terminate immediately, and signals to sendmail that the address has completely resolved. The complete syntax is:

    $#** *mailer***$@ *host***$:***user*

This specifies the {mailer, host, user} 3-tuple necessary to direct the mailer. If the mailer is local the host part may be omitted. The *mailer* and *host* must be a single word, but the *user* may be multi-part.

A RHS may also be preceded by a **$@** or a **$:** to control evaluation. A **$@** prefix causes the ruleset to return with the remainder of the RHS as the value. A **$:** prefix causes the rule to terminate immediately, but the ruleset to continue; this can be used to avoid continued application of a rule. The prefix is stripped before continuing.

The $@ and $: prefixes may preceed a $> spec; for example:

R$+ $:$>7$1

matches anything, passes that to ruleset seven, and continues; the $: is necessary to avoid an infinite loop.

5.2.5. Semantics of rewriting rule sets

There are five rewriting sets that have specific semantics. These are related as depicted by figure 2.

Ruleset three should turn the address into "canonical form." This form should have the basic syntax:

local-part@host-domain-spec

If no "@" sign is specified, then the host-domain-spec *may* be appended from the sender address (if the **C** flag is set in the mailer definition corresponding to the *sending* mailer). Ruleset three is applied by sendmail before doing anything with any address.

Ruleset zero is applied after ruleset three to addresses that are going to actually specify recipients. It must resolve to a {*mailer, host, user*} triple. The *mailer* must be defined in the mailer definitions from the configuration file. The *host* is defined into the **$h** macro for use in the argv expansion of the specified mailer.

Rulesets one and two are applied to all sender and recipient addresses respectively. They are applied before any specification in the mailer definition. They must never resolve.

Ruleset four is applied to all addresses in the message. It is typically used to translate internal to external form.

5.2.6. Mailer flags etc.

There are a number of flags that may be associated with each mailer, each identified by a letter of the alphabet. Many of them are assigned semantics internally. These are detailed in Appendix C. Any other flags may be used freely to conditionally assign headers to messages destined for particular mailers.
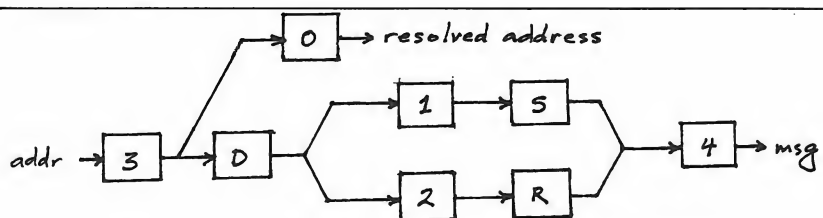
Figure 2 — Rewriting Set Semantics
D—Sender domain addition S—mailer-specific sender rewriting R— mailer-specific recipient rewriting

5.2.7. The "error" mailer

The mailer with the special name "error" can be used to generate a user error. The (optional) host field is a numeric exit status to be returned, and the user field is a message to be printed. For example, the entry:

        $#error$:Host unknown in this domain

on the RHS of a rule will cause the specified error to be generated if the LHS matches. This mailer is only functional in ruleset zero.

5.3. Building a Configuration File From Scratch

Building a configuration table from scratch is an extremely difficult job. Fortunately, it is almost never necessary to do so; nearly every situation that may come up may be resolved by changing an existing table. In any case, it is critical that you understand what it is that you are trying to do and come up with a philosophy for the configuration table. This section is intended to explain what the real purpose of a configuration table is and to give you some ideas for what your philosophy might be.

5.3.1. What you are trying to do

The configuration table has three major purposes. The first and simplest is to set up the environment for *sendmail.* This involves setting the options, defining a few critical macros, etc. Since these are described in other places, we will not go into more detail here.

The second purpose is to rewrite addresses in the message. This should typically be done in two phases. The first phase maps addresses in any format into a canonical form. This should be done in ruleset three. The second phase maps this canonical form into the syntax appropriate for the receiving mailer. *Sendmail* does this in three subphases. Rulesets one and two are applied to all sender and recipient addresses respectively. After this, you may specify per-mailer rulesets for both sender and recipient addresses; this allows mailer-specific customization. Finally, ruleset four is applied to do any default conversion to external form.

The third purpose is to map addresses into the actual set of instructions necessary to get the message delivered. Ruleset zero must resolve to the internal form, which is in turn used as a pointer to a mailer descriptor. The mailer descriptor describes the interface requirements of the mailer.

5.3.2. Philosophy

The particular philosophy you choose will depend heavily on the size and structure of your organization. I will present a few possible philosophies here.

One general point applies to all of these philosophies: it is almost always a mistake to try to do full name resolution. For example, if you are trying to get names of the form "user@host" to the Arpanet, it does not pay to route them to "xyzvax!decvax!ucbvax!c70:user@host" since you then depend on several links not under your control. The best approach to this problem is to simply forward to "xyzvax!user@host" and let xyzvax worry about it from there. In summary, just get the message closer to the destination, rather than determining the full path.

5.3.2.1. Large site, many hosts — minimum information

Berkeley is an example of a large site, i.e., more than two or three hosts. We have decided that the only reasonable philosophy in our environment is to designate one host as the guru for our site. It must be able to resolve any piece of mail it receives. The other sites should have the minimum amount of information they can get away with. In addition, any information they do have should be hints rather than solid information.

For example, a typical site on our local ether network is "monet." Monet has a list of known ethernet hosts; if it receives mail for any of them, it can do direct delivery. If it receives mail for any unknown host, it just passes it directly to "ucbvax," our master host. Ucbvax may determine that the host name is illegal and reject the message, or may be able to do delivery. However, it is important to note that when a new ethernet host is added, the only host that *must* have its tables updated is ucbvax; the others *may* be updated as convenient, but this is not critical.

This picture is slightly muddied due to network connections that are not actually located on ucbvax. For example, our TCP connection is currently on "ucbarpa." However, monet *does not* know about this; the information is hidden totally between ucbvax and ucbarpa. Mail going from monet to a TCP host is transfered via the ethernet from monet to ucbvax, then via the ethernet from ucbvax to ucbarpa, and then is submitted to the Arpanet. Although this involves some extra hops, we feel this is an acceptable tradeoff.

An interesting point is that it would be possible to update monet to send TCP mail directly to ucbarpa if the load got too high; if monet failed to note a host as a TCP host it would go via ucbvax as before, and if monet incorrectly sent a message to ucbarpa it would still be sent by ucbarpa to ucbvax as before. The only problem that can occur is loops, as if ucbarpa thought that ucbvax had the TCP connection and vice versa. For this reason, updates should *always* happen to the master host first.

This philosophy results as much from the need to have a single source for the configuration files (typically built using *m4* (1) or some similar tool) as any logical need. Maintaining more than three separate tables by hand is essentially an impossible job.

5.3.2.2. Small site — complete information

A small site (two or three hosts) may find it more reasonable to have complete information at each host. This would require that each host know exactly where each network connection is, possibly including the names of each host on that network. As long as the site remains small and the the configuration remains relatively static, the update problem will probably not be too great.

5.3.2.3. Single host

This is in some sense the trivial case. The only major issue is trying to insure that you don't have to know too much about your environment. For example, if you have a UUCP connection you might find it useful to know about the names of hosts connected directly to you, but this is really not necessary since this may be determined from the syntax.

5.3.3. Relevant Issues

The canonical form you use should almost certainly be as specified in the Arpanet protocols RFC819 and RFC822. Copies of these RFC's are included on the *sendmail* tape as *doc/rfc819.lpr* and *doc/rfc822.lpr.*

RFC822 describes the format of the mail message itself. *Sendmail* follows this RFC closely, to the extent that many of the standards described in this document can not be changed without changing the code. In particular, the following characters have special interpretations:

< > ( ) " \

Any attempt to use these characters for other than their RFC822 purpose in addresses is probably doomed to disaster.

RFC819 describes the specifics of the domain-based addressing. This is touched on in RFC822 as well. Essentially each host is given a name which is a right-to-left dot qualified pseudo-path from a distinguished root. The elements of the path need not be physical hosts; the domain is logical rather than physical. For example, at Berkeley one legal host is "a.cc.berkeley.arpa"; reading from right to left, "arpa" is a top level domain (related to, but not limited to, the physical Arpanet), "berkeley" is both an Arpanet host and a logical domain which is actually interpreted by a host called ucbvax (which is actually just the "major" host for this domain), "cc" represents the Computer Center, (in this case a strictly logical entity), and "a" is a host in the Computer Center; this particular host happens to be connected via berknet, but other hosts might be connected via one of two ethernets or some other network.

Beware when reading RFC819 that there are a number of errors in it.

5.3.4. How to proceed

Once you have decided on a philosophy, it is worth examining the available configuration tables to decide if any of them are close enough to steal major parts of. Even under the worst of conditions, there is a fair amount of boiler plate that can be collected safely.

The next step is to build ruleset three. This will be the hardest part of the job. Beware of doing too much to the address in this ruleset, since anything you do will reflect through to the message. In particular, stripping of local domains is best deferred, since this can leave you with addresses with no domain spec at all. Since *sendmail* likes to append the sending domain to addresses with no domain, this can change the semantics of addresses. Also try to avoid fully qualifying domains in this ruleset. Although technically legal, this can lead to unpleasantly and unnecessarily long addresses reflected into messages. The Berkeley configuration files define ruleset nine to qualify domain names and strip local domains. This is called from ruleset zero to get all addresses into a cleaner form.

Once you have ruleset three finished, the other rulesets should be relatively trivial. If you need hints, examine the supplied configuration tables.

5.3.5. Testing the rewriting rules — the —bt flag

When you build a configuration table, you can do a certain amount of testing using the "test mode" of *sendmail*. For example, you could invoke *sendmail* as:

sendmail —bt —Ctest.cf

which would read the configuration file "test.cf" and enter test mode. In this mode, you enter lines of the form:

rwset address

where *rwset* is the rewriting set you want to use and *address* is an address to apply the set to. Test mode shows you the steps it takes as it proceeds, finally showing you the address it ends up with. You may use a comma separated list of rwsets for sequential application of rules to an input; ruleset three is always applied first. For example:

1,21,4 monet:bollard

first applies ruleset three to the input "monet:bollard." Ruleset one is then applied to the output of ruleset three, followed similarly by rulesets twenty-one and four.

If you need more detail, you can also use the "—d21" flag to turn on more debugging. For example,

sendmail —bt —d21.99

turns on an incredible amount of information; a single word address is probably going

to print out several pages worth of information.

5.3.6. Building mailer descriptions

To add an outgoing mailer to your mail system, you will have to define the characteristics of the mailer.

Each mailer must have an internal name. This can be arbitrary, except that the names "local" and "prog" must be defined.

The pathname of the mailer must be given in the P field. If this mailer should be accessed via an IPC connection, use the string "[IPC]" instead.

The F field defines the mailer flags. You should specify an "f" or "r" flag to pass the name of the sender as a −f or −r flag respectively. These flags are only passed if they were passed to *sendmail,* so that mailers that give errors under some circumstances can be placated. If the mailer is not picky you can just specify "−f $g" in the argv template. If the mailer must be called as **root** the "S" flag should be given; this will not reset the userid before calling the mailer[3]. If this mailer is local (i.e., will perform final delivery rather than another network hop) the "l" flag should be given. Quote characters (backslashes and " marks) can be stripped from addresses if the "s" flag is specified; if this is not given they are passed through. If the mailer is capable of sending to more than one user on the same host in a single transaction the "m" flag should be stated. If this flag is on, then the argv template containing $u will be repeated for each unique user on a given host. The "e" flag will mark the mailer as being "expensive," which will cause *sendmail* to defer connection until a queue run[4].

An unusual case is the "C" flag. This flag applies to the mailer that the message is received from, rather than the mailer being sent to; if set, the domain spec of the sender (i.e., the "@host.domain" part) is saved and is appended to any addresses in the message that do not already contain a domain spec. For example, a message of the form:

> From: eric@ucbarpa
> To: wnj@monet, mckusick

will be modified to:

> From: eric@ucbarpa
> To: wnj@monet, mckusick@ucbarpa

if and only if the "C" flag is defined in the mailer corresponding to "eric@ucbarpa."

Other flags are described in Appendix C.

The S and R fields in the mailer description are per-mailer rewriting sets to be applied to sender and recipient addresses respectively. These are applied after the sending domain is appended and the general rewriting sets (numbers one and two) are applied, but before the output rewrite (ruleset four) is applied. A typical use is to append the current domain to addresses that do not already have a domain. For example, a header of the form:

> From: eric

might be changed to be:

> From: eric@ucbarpa

or

---

[3]*Sendmail* must be running setuid to root for this to work.

[4]The "c" configuration option must be given for this to be effective.

From: ucbvax!eric

depending on the domain it is being shipped into. These sets can also be used to do special purpose output rewriting in cooperation with ruleset four.

The E field defines the string to use as an end-of-line indication. A string containing only newline is the default. The usual backslash escapes (\r, \n, \f, \b) may be used.

Finally, an argv template is given as the E field. It may have embedded spaces. If there is no argv with a $u macro in it, *sendmail* will speak SMTP to the mailer. If the pathname for this mailer is "[IPC]," the argv should be

IPC $h [*port*]

where *port* is the optional port number to connect to.

For example, the specifications:

Mlocal, P=/bin/mail, F=rlsm S=10, R=20, A=mail −d $u
Mether,P=[IPC],   F=meC, S=11, R=21, A=IPC $h, M=100000

specifies a mailer to do local delivery and a mailer for ethernet delivery. The first is called "local," is located in the file "/bin/mail," takes a picky −r flag, does local delivery, quotes should be stripped from addresses, and multiple users can be delivered at once; ruleset ten should be applied to sender addresses in the message and ruleset twenty should be applied to recipient addresses; the argv to send to a message will be the word "mail," the word "−d," and words containing the name of the receiving user. If a −r flag is inserted it will be between the words "mail" and "−d." The second mailer is called "ether," it should be connected to via an IPC connection, it can handle multiple users at once, connections should be deferred, and any domain from the sender address should be appended to any receiver name without a domain; sender addresses should be processed by ruleset eleven and recipient addresses by ruleset twenty-one. There is a 100,000 byte limit on messages passed through this mailer.

APPENDIX A

COMMAND LINE FLAGS

Arguments must be presented with flags before addresses. The flags are:

−f *addr*   The sender's machine address is *addr*. This flag is ignored unless the real user is listed as a "trusted user" or if *addr* contains an exclamation point (because of certain restrictions in UUCP).

−r *addr*   An obsolete form of −f.

−h *cnt*   Sets the "hop count" to *cnt*. This represents the number of times this message has been processed by *sendmail* (to the extent that it is supported by the underlying networks). *Cnt* is incremented during processing, and if it reaches MAX-HOP (currently 30) *sendmail* throws away the message with an error.

−F*name*   Sets the full name of this user to *name*.

−n   Don't do aliasing or forwarding.

−t   Read the header for "To:", "Cc:", and "Bcc:" lines, and send to everyone listed in those lists. The "Bcc:" line will be deleted before sending. Any addresses in the argument vector will be deleted from the send list.

−b*x*   Set operation mode to *x*. Operation modes are:

        m   Deliver mail (default)
        a   Run in arpanet mode (see below)
        s   Speak SMTP on input side
        d   Run as a daemon
        t   Run in test mode
        v   Just verify addresses, don't collect or deliver
        i   Initialize the alias database
        p   Print the mail queue
        z   Freeze the configuration file

    The special processing for the ARPANET includes reading the "From:" line from the header to find the sender, printing ARPANET style messages (preceded by three digit reply codes for compatibility with the FTP protocol [Neigus73, Postel74, Postel77]), and ending lines of error messages with <CRLF>.

−q*time*   Try to process the queued up mail. If the time is given, a sendmail will run through the queue at the specified interval to deliver queued mail; otherwise, it only runs once.

−C*file*   Use a different configuration file.

−d*level*   Set debugging level.

−o*x value*   Set option *x* to the specified *value*. These options are described in Appendix B.

There are a number of options that may be specified as primitive flags (provided for compatibility with *delivermail*). These are the e, i, m, and v options. Also, the f option may be specified as the −s flag.

APPENDIX B

CONFIGURATION OPTIONS

The following options may be set using the **−o** flag on the command line or the **O** line in the configuration file:

A*file*   Use the named *file* as the alias file. If no file is specified, use *aliases* in the current directory.

a   If set, wait for an "@:@" entry to exist in the alias database before starting up. If it does not appear in five minutes, rebuild the database.

c   If an outgoing mailer is marked as being expensive, don't connect immediately. This requires that queueing be compiled in, since it will depend on a queue run process to actually send the mail.

d*x*   Deliver in mode *x*. Legal modes are:

      i   Deliver interactively (synchronously)
      b   Deliver in background (asynchronously)
      q   Just queue the message (deliver during queue run)

D   If set, rebuild the alias database if necessary and possible. If this option is not set, *sendmail* will never rebuild the alias database unless explicitly requested using **−bi**.

e*x*   Dispose of errors using mode *x*. The values for *x* are:

      p   Print error messages (default)
      q   No messages, just give exit status
      m   Mail back errors
      w   Write back errors (mail if user not logged in)
      e   Mail back errors and give zero exit stat always

F*n*   The temporary file mode, in octal. 644 and 600 are good choices.

f   Save Unix-style "From" lines at the front of headers. Normally they are assumed redundant and discarded.

g*n*   Set the default group id for mailers to run in to *n*.

H*file*   Specify the help file for SMTP.

i   Ignore dots in incoming messages.

L*n*   Set the default log level to *n*.

M*x value*   Set the macro *x* to *value*. This is intended only for use from the command line.

m   Send to me too, even if I am in an alias expansion.

o   Assume that the headers may be in old format, i.e., spaces delimit names. This actually turns on an adaptive algorithm: if any recipient address contains a comma, parenthesis, or angle bracket, it will be assumed that commas already exist. If this flag is not on, only commas delimit names. Headers are always output with commas between the names.

Q*dir*   Use the named *dir* as the queue directory.

r*time*   Timeout reads after *time* interval.

S*file*	.	Log statistics in the named *file*.
s		Be super-safe when running things, i.e., always instantiate the queue file, even if you are going to attempt immediate delivery. *Sendmail* always instantiates the queue file before returning control the the client under any circumstances.
T*time*		Set the queue timeout to *time*. After this interval, messages that have not been successfully sent will be returned to the sender.
t*S,D*		Set the local timezone name to *S* for standard time and *D* for daylight time; this is only used under version six.
u*n*		Set the default userid for mailers to *n*. Mailers without the *S* flag in the mailer definition will run as this user.
v		Run in verbose mode.

APPENDIX C

MAILER FLAGS

The following flags may be set in the mailer description.

f   The mailer wants a −f *from* flag, but only if this is a network forward operation (i.e., the mailer will give an error if the executing user does not have special permissions).

r   Same as f, but sends a −r flag.

S   Don't reset the userid before calling the mailer. This would be used in a secure environment where *sendmail* ran as root. This could be used to avoid forged addresses. This flag is suppressed if given from an "unsafe" environment (e.g, a user's mail.cf file).

n   Do not insert a UNIX-style "From" line on the front of the message.

l   This mailer is local (i.e., final delivery will be performed).

s   Strip quote characters off of the address before calling the mailer.

m   This mailer can send to multiple users on the same host in one transaction. When a $u macro occurs in the *argv* part of the mailer definition, that field will be repeated as necessary for all qualifying users.

F   This mailer wants a "From:" header line.

D   This mailer wants a "Date:" header line.

M   This mailer wants a "Message-Id:" header line.

x   This mailer wants a "Full-Name:" header line.

P   This mailer wants a "Return-Path:" line.

u   Upper case should be preserved in user names for this mailer.

h   Upper case should be preserved in host names for this mailer.

A   This is an Arpanet-compatible mailer, and all appropriate modes should be set.

U   This mailer wants Unix-style "From" lines with the ugly UUCP-style "remote from <host>" on the end.

e   This mailer is expensive to connect to, so try to avoid connecting normally; any necessary connection will occur during a queue run.

X   This mailer want to use the hidden dot algorithm as specified in RFC821; basically, any line beginning with a dot will have an extra dot prepended (to be stripped at the other end). This insures that lines in the message containing a dot will not terminate the message prematurely.

L   Limit the line lengths as specified in RFC821.

P   Use the return-path in the SMTP "MAIL FROM:" command rather than just the return address; although this is required in RFC821, many hosts do not process return paths properly.

I   This mailer will be speaking SMTP to another *sendmail* — as such it can use special protocol features. This option is not required (i.e., if this option is omitted the transmission will still operate successfully, although perhaps not as efficiently as possible).

C   If mail is *received* from a mailer with this flag set, any addresses in the header that do not have an at sign ("@") after being rewritten by ruleset three will have the "@domain" clause from the sender tacked on. This allows mail with headers of the form:

    From: usera@hosta
    To: userb@hostb, userc

to be rewritten as:

    From: usera@hosta
    To: userb@hostb, userc@hosta

automatically.

APPENDIX D

OTHER CONFIGURATION

There are some configuration changes that can be made by recompiling *sendmail*. These are located in three places:

md/config.m4   These contain operating-system dependent descriptions. They are interpolated into the Makefiles in the *src* and *aux* directories. This includes information about what version of UNIX you are running, what libraries you have to include, etc.

src/conf.h   Configuration parameters that may be tweaked by the installer are included in conf.h.

src/conf.c   Some special routines and a few variables may be defined in conf.c. For the most part these are selected from the settings in conf.h.

Parameters in md/config.m4

The following compilation flags may be defined in the *m4CONFIG* macro in *md/config.m4* to define the environment in which you are operating.

V6   If set, this will compile a version 6 system, with 8-bit user id's, single character tty id's, etc.

VMUNIX   If set, you will be assumed to have a Berkeley 4BSD or 4.1BSD, including the *vfork*(2) system call, special types defined in <sys/types.h> (e.g, u_char), etc.

If none of these flags are set, a version 7 system is assumed.

You will also have to specify what libraries to link with *sendmail* in the *m4LIBS* macro. Most notably, you will have to include if you are running a 4.1BSD system.

Parameters in src/conf.h

Parameters and compilation options are defined in conf.h. Most of these need not normally be tweaked; common parameters are all in sendmail.cf. However, the sizes of certain primitive vectors, etc., are included in this file. The numbers following the parameters are their default value.

MAXLINE [256]   The maximum line length of any input line. If message lines exceed this length they will still be processed correctly; however, header lines, configuration file lines, alias lines, etc., must fit within this limit.

MAXNAME [128] The maximum length of any name, such as a host or a user name.

MAXFIELD [2500]
  The maximum total length of any header field, including continuation lines.

MAXPV [40]   The maximum number of parameters to any mailer. This limits the number of recipients that may be passed in one transaction.

MAXHOP [30]   When a message has been processed more than this number of times, sendmail rejects the message on the assumption that there has been an aliasing loop. This can be determined from the −h flag or by counting the number of trace fields (i.e, "Received:" lines) in the message header.

MAXATOM [100] The maximum number of atoms (tokens) in a single address. For example, the address "eric@Berkeley" is three atoms.

MAXMAILERS [25]
> The maximum number of mailers that may be defined in the configuration file.

MAXRWSETS [30]
> The maximum number of rewriting sets that may be defined.

MAXPRIORITIES [25]
> The maximum number of values for the "Precedence:" field that may be defined (using the **P** line in sendmail.cf).

MAXTRUST [30] The maximum number of trusted users that may be defined (using the **T** line in sendmail.cf).

A number of other compilation options exist. These specify whether or not specific code should be compiled in.

DBM
> If set, the "DBM" package in UNIX is used (see DBM(3X) in [UNIX80]). If not set, a much less efficient algorithm for processing aliases is used.

DEBUG
> If set, debugging information is compiled in. To actually get the debugging output, the **−d** flag must be used.

LOG
> If set, the *syslog* routine in use at some sites is used. This makes an informational log record for each message processed, and makes a higher priority log record for internal system errors.

QUEUE
> This flag should be set to compile in the queueing code. If this is not set, mailers must accept the mail immediately or it will be returned to the sender.

SMTP
> If set, the code to handle user and server SMTP will be compiled in. This is only necessary if your machine has some mailer that speaks SMTP.

DAEMON
> If set, code to run a daemon is compiled in. This code is for 4.2BSD if the NVMUNIX flag is specified; otherwise, 4.1a BSD code is used. Beware however that there are bugs in the 4.1a code that make it impossible for **sendmail** to work correctly under heavy load.

UGLYUUCP
> If you have a UUCP host adjacent to you which is not running a reasonable version of *rmail*, you will have to set this flag to include the "remote from sysname" info on the from line. Otherwise, UUCP gets confused about where the mail came from.

NOTUNIX
> If you are using a non-UNIX mail format, you can set this flag to turn off special processing of UNIX-style "From " lines.

Configuration in src/conf.c

Not all header semantics are defined in the configuration file. Header lines that should only be included by certain mailers (as well as other more obscure semantics) must be specified in the *HdrInfo* table in *conf.c*. This table contains the header name (which should be in all lower case) and a set of header control flags (described below), The flags are:

H_ACHECK
> Normally when the check is made to see if a header line is compatible with a mailer, *sendmail* will not delete an existing line. If this flag is set, *sendmail* will delete even existing header lines. That is, if this bit is set and the mailer does not have flag bits set that intersect with the required mailer flags in the header definition in sendmail.cf, the header line is *always* deleted.

H_EOH
> If this header field is set, treat it like a blank line, i.e., it will signal the end of the header and the beginning of the message text.

H_FORCE
> Add this header entry even if one existed in the message before. If a header entry does not have this bit set, *sendmail* will not add another header line if a header line of this name already existed. This would normally be used to stamp the message by everyone who handled it.

H_TRACE   If set, this is a timestamp (trace) field. If the number of trace fields in a message exceeds a preset amount the message is returned on the assumption that it has an aliasing loop.

H_RCPT   If set, this field contains recipient addresses. This is used by the —t flag to determine who to send to when it is collecting recipients from the message.

H_FROM   This flag indicates that this field specifies a sender. The order of these fields in the *HdrInfo* table specifies *sendmail's* preference for which field to return error messages to.

Let's look at a sample *HdrInfo* specification:

```
struct hdrinfo          HdrInfo[] =
{
        /* originator fields, most to least significant */
    "resent-sender",    H_FROM,
    "resent-from",      H_FROM,
    "sender",           H_FROM,
    "from",             H_FROM,
    "full-name",        H_ACHECK,
        /* destination fields */
    "to",               H_RCPT,
    "resent-to",        H_RCPT,
    "cc",               H_RCPT,
        /* message identification and control */
    "message",          H_EOH,
    "text",             H_EOH,
        /* trace fields */
    "received",         H_TRACE|H_FORCE,

    NULL,               0,
};
```

This structure indicates that the "To:", "Resent-To:", and "Cc:" fields all specify recipient addresses. Any "Full-Name:" field will be deleted unless the required mailer flag (indicated in the configuration file) is specified. The "Message:" and "Text:" fields will terminate the header; these are specified in new protocols [NBS80] or used by random dissenters around the network world. The "Received:" field will always be added, and can be used to trace messages.

There are a number of important points here. First, header fields are not added automatically just because they are in the *HdrInfo* structure; they must be specified in the configuration file in order to be added to the message. Any header fields mentioned in the configuration file but not mentioned in the *HdrInfo* structure have default processing performed; that is, they are added unless they were in the message already. Second, the *HdrInfo* structure only specifies cliched processing; certain headers are processed specially by ad hoc code regardless of the status specified in *HdrInfo*. For example, the "Sender:" and "From:" fields are always scanned on ARPANET mail to determine the sender; this is used to perform the "return to sender" function. The "From:" and "Full-Name:" fields are used to determine the full name of the sender if possible; this is stored in the macro $x and used in a number of ways.

The file *conf.c* also contains the specification of ARPANET reply codes. There are four classifications these fall into:

```
char Arpa_Info[] =      "050"; /* arbitrary info */
char Arpa_TSyserr[] =   "455"; /* some (transient) system error */
char Arpa_PSyserr[] =   "554"; /* some (transient) system error */
char Arpa_Usrerr[] =    "554"; /* some (fatal) user error */
```

The class *Arpa_Info* is for any information that is not required by the protocol, such as forwarding information. *Arpa_TSyserr* and *Arpa_PSyserr* is printed by the *syserr* routine. TSyserr is

printed out for transient errors, whereas PSyserr is printed for permanent errors; the distinction is made based on the value of *errno*. Finally, *Arpa_Usrerr* is the result of a user error and is generated by the *usrerr* routine; these are generated when the user has specified something wrong, and hence the error is permanent, i.e., it will not work simply by resubmitting the request.

If it is necessary to restrict mail through a relay, the *checkcompat* routine can be modified. This routine is called for every recipient address. It can return **TRUE** to indicate that the address is acceptable and mail processing will continue, or it can return **FALSE** to reject the recipient. If it returns false, it is up to *checkcompat* to print an error message (using *usrerr*) saying why the message is rejected. For example, *checkcompat* could read:

```
bool
checkcompat(to)
    register ADDRESS *to;
{
    if (MsgSize > 50000 && to->q_mailer != LocalMailer)
    {
        usrerr("Message too large for non-local delivery");
        NoReturn = TRUE;
        return (FALSE);
    }
    return (TRUE);
}
```

This would reject messages greater than 50000 bytes unless they were local. The *NoReturn* flag can be sent to supress the return of the actual body of the message in the error return. The actual use of this routine is highly dependent on the implementation, and use should be limited.

APPENDIX E

SUMMARY OF SUPPORT FILES

This is a summary of the support files that *sendmail* creates or generates.

/usr/lib/sendmail
> The binary of *sendmail.*

/usr/bin/newaliases
> A link to /usr/lib/sendmail; causes the alias database to be rebuilt. Running this program is completely equivalent to giving *sendmail* the **-bi** flag.

/usr/bin/mailq Prints a listing of the mail queue. This program is equivalent to using the **-bp** flag to *sendmail.*

/usr/lib/sendmail.cf
> The configuration file, in textual form.

/usr/lib/sendmail.fc
> The configuration file represented as a memory image.

/usr/lib/sendmail.hf
> The SMTP help file.

/usr/lib/sendmail.st
> A statistics file; need not be present.

/usr/lib/aliases The textual version of the alias file.

/usr/lib/aliases.{pag,dir}
> The alias file in *dbm* (3) format.

/etc/syslog   The program to do logging.

/etc/syslog.conf The configuration file for syslog.

/etc/syslog.pid Contains the process id of the currently running syslog.

/usr/spool/mqueue
> The directory in which the mail queue and temporary files reside.

/usr/spool/*mqueue*/qf*
> Control (queue) files for messages.

/usr/spool/mqueue/df*
> Data files.

/usr/spool/mqueue/lf*
> Lock files

/usr/spool/mqueue/tf*
> Temporary versions of the qf files, used during queue file rebuild.

/usr/spool/mqueue/nf*
> A file used when creating a unique id.

/usr/spool/mqueue/xf*
> A transcript of the current session.

4.2BSD Line Printer Spooler Manual

Revised July 27, 1983

Ralph Campbell

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

(415) 642-7780

ABSTRACT

This document describes the structure and installation procedure for the line printer spooling system developed for the 4.2BSD version of the UNIX* operating system.

1. Overview

The line printer system supports:

- multiple printers,
- multiple spooling queues,
- both local and remote printers, and
- printers attached via serial lines which require line initialization such as the baud rate.

Raster output devices such as a Varian or Versatec, and laser printers such as an Imagen, are also supported by the line printer system.

The line printer system consists mainly of the following files and commands:

/etc/printcap	printer configuration and capability data base
/usr/lib/lpd	line printer daemon, does all the real work
/usr/ucb/lpr	program to enter a job in a printer queue
/usr/ucb/lpq	spooling queue examination program
/usr/ucb/lprm	program to delete jobs from a queue
/etc/lpc	program to administer printers and spooling queues
/dev/printer	socket on which lpd listens

The file /etc/printcap is a master data base describing line printers directly attached to a machine and, also, printers accessible across a network. The manual page entry *printcap*(5) provides the ultimate definition of the format of this data base, as well as indicating default values for important items such as the directory in which spooling is performed. This document highlights the important information which may be placed *printcap*.

---

* UNIX is a trademark of Bell Laboratories.

2. Commands

2.1. lpd — line printer dameon

The program *lpd*(8), usually invoked at boot time from the /etc/rc file, acts as a master server for coordinating and controlling the spooling queues configured in the printcap file. When *lpd* is started it makes a single pass through the *printcap* database restarting any printers which have jobs. In normal operation *lpd* listens for service requests on multiple sockets, one in the UNIX domain (named "/dev/printer") for local requests, and one in the Internet domain (under the "printer" service specification) for requests for printer access from off machine; see *socket*(2) and *services*(5) for more information on sockets and service specifications, respectively. *Lpd* spawns a copy of itself to process the request; the master daemon continues to listen for new requests.

Clients communicate with *lpd* using a simple transaction oriented protocol. Authentication of remote clients is done based on the "privilege port" scheme employed by *rshd*(8C) and *rcmd*(3X). The following table shows the requests understood by *lpd*. In each request the first byte indicates the "meaning" of the request, followed by the name of the printer to which it should be applied. Additional qualifiers may follow, depending on the request.

Request	Interpretation
^Aprinter\n	check the queue for jobs and print any found
^Bprinter\n	receive and queue a job from another machine
^Cprinter [users ...] [jobs ...]\n	return short list of current queue state
^Dprinter [users ...] [jobs ...]\n	return long list of current queue state
^Eprinter person [users ...] [jobs ...]\n	remove jobs from a queue

The *lpr*(1) command is used by users to enter a print job in a local queue and to notify the local *lpd* that there are new jobs in the spooling area. *Lpd* either schedules the job to be printed locally, or in the case of remote printing, attempts to forward the job to the appropriate machine. If the printer cannot be opened or the destination machine is unreachable, the job will remain queued until it is possible to complete the work.

2.2. lpq — show line printer queue

The *lpq*(1) program works recursively backwards displaying the queue of the machine with the printer and then the queue(s) of the machine(s) that lead to it. *Lpq* has two forms of output: in the default, short, format it gives a single line of output per queued job; in the long format it shows the list of files, and their sizes, which comprise a job.

2.3. lprm — remove jobs from a queue

The *lprm*(1) command deletes jobs from a spooling queue. If necessary, *lprm* will first kill off a running daemon which is servicing the queue, restarting it after the required files are removed. When removing jobs destined for a remote printer, *lprm* acts similarly to *lpq* except it first checks locally for jobs to remove and then tries to remove files in queues off-machine.

2.4. lpc — line printer control program

The *lpc*(8) program is used by the system administrator to control the operation of the line printer system. For each line printer configured in /etc/printcap, *lpc* may be used to:

● disable or enable a printer,

● disable or enable a printer's spooling queue,

● rearrange the order of jobs in a spooling queue,

● find the status of printers, and their associated spooling queues and printer dameons.

3. Access control

The printer system maintains protected spooling areas so that users cannot circumvent printer accounting or remove files other than their own. The strategy used to maintain protected spooling areas is as follows:

- The spooling area is writable only by a *daemon* user and *spooling* group.

- The *lpr* program runs setuid *root* and setgid *spooling*. The *root* access is used to read any file required, verifying accessibility with an *access*(2) call. The group ID is used in setting up proper ownership of files in the spooling area for *lprm*.

- Control files in a spooling area are made with *daemon* ownership and group ownership *spooling*. Their mode is 0660. This insures control files are not modified by a user and that no user can remove files except through *lprm*.

- The spooling programs, *lpd*, *lpq*, and *lprm* run setuid *root* and setgid *spooling* to access spool files and printers.

- The printer server, *lpd*, uses the same verification procedures as *rshd*(8C) in authenticating remote clients. The host on which a client resides must be present in the file /etc/hosts.equiv, used to create clusters of machines under a single administration.

In practice, none of *lpd*, *lpq*, or *lprm* would have to run as user *root* if remote spooling were not supported. In previous incarnations of the printer system *lpd* ran setuid *daemon*, setgid *spooling*, and *lpq* and *lprm* ran setgid *spooling*.

4. Setting up

The 4.2BSD release comes with the necessary programs installed and with the default line printer queue created. If the system must be modified, the makefile in the directory /usr/src/usr.lib/lpr should be used in recompiling and reinstalling the necessary programs.

The real work in setting up is to create the *printcap* file and any printer filters for printers not supported in the distribution system.

4.1. Creating a printcap file

The *printcap* database contains one or more entries per printer. A printer should have a separate spooling directory; otherwise, jobs will be printed on different printers depending on which printer daemon starts first. This section describes how to create entries for printers which do not conform to the default printer description (an LP-11 style interface to a standard, band printer).

4.1.1. Printers on serial lines

When a printer is connected via a serial communication line it must have the proper baud rate and terminal modes set. The following example is for a DecWriter III printer connected locally via a 1200 baud serial line.

```
lp|LA-180 DecWriter III:\
    :lp=/dev/lp:br#1200:fs#06320:\
    :tr=\f:of=/usr/lib/lpf:lf=/usr/adm/lpd-errs:
```

The **lp** entry specifies the file name to open for output. In this case it could be left out since "/dev/lp" is the default. The **br** entry sets the baud rate for the tty line and the **fs** entry sets CRMOD, no parity, and XTABS (see *tty*(4)). The **tr** entry indicates a form-feed should be printed when the queue empties so the paper can be torn off without turning the printer off-line and pressing form feed. The **of** entry specifies the filter program *lpf* should be used for printing the files; more will be said about filters later. The last entry causes errors to be written to the file "/usr/adm/lpd-errs" instead of the console.

4.1.2. Remote printers

Printers which reside on remote hosts should have an empty **lp** entry. For example, the following printcap entry would send output to the printer named "lp" on the machine "ucbvax".

```
lp|default line printer:\
    :lp=:rm=ucbvax:rp=lp:sd=/usr/spool/vaxlpd:
```

The **rm** entry is the name of the remote machine to connect to; this name must appear in the /etc/hosts database, see *hosts*(5). The **rp** capability indicates the *name* of the printer on the remote machine is "lp"; in this case it could be left out since this is the default value. The **sd** entry specifies "/usr/spool/vaxlpd" as the spooling directory instead of the default value of "/usr/spool/lpd".

4.2. Output filters

Filters are used to handle device dependencies and to perform accounting functions. The output filter **of** is used to filter text data to the printer device when accounting is not used or when all text data must be passed through a filter. It is not intended to perform accounting since it is started only once, all text files are filtered through it, and no provision is made for passing owners' login name, identifying the begining and ending of jobs, etc. The other filters (if specified) are started for each file printed and perform accounting if there is an **af** entry. If entries for both **of** and one of the other filters are specified, the output filter is used only to print the banner page; it is then stopped to allow other filters access to the printer. An example of a printer which requires output filters is the Benson-Varian.

```
va|varian|Benson-Varian:\
    :lp=/dev/va0:sd=/usr/spool/vad:of=/usr/lib/vpf:\
    :tf=/usr/lib/rvcat:mx#2000:pl#58:tr=\f:
```

The **tf** entry specifies "/usr/lib/rvcat" as the filter to be used in printing *troff*(1) output. This filter is needed to set the device into print mode for text, and plot mode for printing *troff* files and raster images (see *va*(4V)). Note that the page length is set to 58 lines by the **pl** entry for 8.5" by 11" fan-fold paper. To enable accounting, the varian entry would be augmented with an **af** filter as shown below.

```
va|varian|Benson-Varian:\
    :lp=/dev/va0:sd=/usr/spool/vad:of=/usr/lib/vpf:\
    :if=/usr/lib/vpf:tf=/usr/lib/rvcat:af=/usr/adm/vaacct:\
    :mx#2000:pl#58:tr=\f:
```

5. Output filter specifications

The filters supplied with 4.2BSD handle printing and accounting for most common line printers, the Benson-Varian, the wide (36") and narrow (11") Versatec printer/plotters. For other devices or accounting methods, it may be necessary to create a new filter.

Filters are spawned by *lpd* with their standard input the data to be printed, and standard output the printer. The standard error is attached to the **lf** file for logging errors. A filter must return a 0 exit code if there were no errors, 1 if the job should be reprinted, and 2 if the job should be thrown away. When *lprm* sends a kill signal to the *lpd* process controlling printing, it sends a SIGINT signal to all filters and descendents of filters. This signal can be trapped by filters which need to perform cleanup operations such as deleting temporary files.

Arguments passed to a filter depend on its type. The **of** filter is called with the following arguments.

```
ofiler −wwidth −llength
```

The *width* and *length* values come from the **pw** and **pl** entries in the printcap database. The **if**

filter is passed the following parameters.

filter [−c] −wwidth −llength −iindent −n login −h host accounting_file

The −c flag is optional, and only supplied when control characters are to be passed uninterpreted to the printer (when the −l option of *lpr* is used to print the file). The −w and −l parameters are the same as for the **of** filter. The −n and −h parameters specify the login name and host name of the job owner. The last argument is the name of the accounting file from *printcap*.

All other filters are called with the following arguments:

filter −xwidth −ylength −n login −h host accounting_file

The −x and −y options specify the horizontal and vertical page size in pixels (from the **px** and **py** entries in the printcap file). The rest of the arguments are the same as for the **if** filter.

6. Line printer Administration

The *lpc* program provides local control over line printer activity. The major commands and their intended use will be described. The command format and remaining commands are described in *lpc*(8).

abort and start

Abort terminates an active spooling daemon on the local host immediately and then disables printing (preventing new daemons from being started by *lpr*). This is normally used to forciblly restart a hung line printer daemon (i.e., *lpq* reports that there is a daemon present but nothing is happening). It does not remove any jobs from the queue (use the *lprm* command instead). *Start* enables printing and requests *lpd* to start printing jobs.

enable and disable

Enable and *disable* allow spooling in the local queue to be turned on/off. This will allow/prevent *lpr* from putting new jobs in the spool queue. It is frequently convenient to turn spooling off while testing new line printer filters since the *root* user can still use *lpr* to put jobs in the queue but no one else can. The other main use is to prevent users from putting jobs in the queue when the printer is expected to be unavailable for a long time.

restart

Restart allows ordinary users to restart printer daemons when *lpq* reports that there is no daemon present.

stop

Stop is used to halt a spooling daemon after the current job completes; this also disables printing. This is a clean way to shutdown a printer in order to perform maintenence, etc. Note that users can still enter jobs in a spool queue while a printer is *stopped*.

topq

Topq places jobs at the top of a printer queue. This can be used to reorder high priority jobs since *lpr* only only provides first-come-first-serve ordering of jobs.

7. Troubleshooting

There are a number of messages which may be generated by the the line printer system. This section categorizes the most common and explains the cause for their generation. Where the message indicates a failure, directions are given to remedy the problem.

In the examples below, the name *printer* is the name of the printer. This would be one of the names from the *printcap* database.

7.1. LPR

lpr: *printer*: **unknown printer**

The *printer* was not found in the *printcap* database. Usually this is a typing mistake; however, it may indicate a missing or incorrect entry in the /etc/printcap file.

lpr: *printer*: **jobs queued, but cannot start daemon.**

The connection to *lpd* on the local machine failed. This usually means the printer server started at boot time has died or is hung. Check the local socket /dev/printer to be sure it still exists (if it does not exist, there is no *lpd* process running). Use

        % ps ax | fgrep lpd

to get a list of process identifiers of running lpd's. The *lpd* to kill is the one which is not listed in any of the "lock" files (the lock file is contained in the spool directory of each printer). Kill the master daemon using the following command.

        % kill *pid*

Then remove /dev/printer and restart the daemon (and printer) with the following commands.

        % rm /dev/printer
        % /usr/lib/lpd

Another possibility is that the *lpr* program is not setuid *root*, setgid *spooling*. This can be checked with

        % ls −lg /usr/ucb/lpr

lpr: *printer*: **printer queue is disabled**

This means the queue was turned off with

        % lpc disable *printer*

to prevent *lpr* from putting files in the queue. This is normally done by the system manager when a printer is going to be down for a long time. The printer can be turned back on by a super-user with *lpc*.

7.2. LPQ

waiting for *printer* **to become ready (offline ?)**

The printer device could not be opened by the daemon. This can happen for a number of reasons, the most common being that the printer is turned off-line. This message can also be generated if the printer is out of paper, the paper is jammed, etc. The actual reason is dependent on the meaning of error codes returned by system device driver. Not all printers supply sufficient information to distinguish when a printer is off-line or having trouble (e.g. a printer connected through a serial line). Another possible cause of this message is some other process, such as an output filter, has an exclusive open on the device. Your only recourse here is to kill off the offending program(s) and restart the printer with *lpc*.

printer **is ready and printing**

The *lpq* program checks to see if a daemon process exists for *printer* and prints the file *status*. If the daemon is hung, a super user can use *lpc* to abort the current daemon and start a new one.

waiting for *host* **to come up**

This indicates there is a daemon trying to connect to the remote machine named *host* in order to send the files in the local queue. If the remote machine is up, *lpd* on the remote machine is probably dead or hung and should be restarted as mentioned for *lpr.*

sending to *host*

The files should be in the process of being transferred to the remote *host.* If not, the local daemon should be aborted and started with *lpc.*

Warning: *printer* **is down**

The printer has been marked as being unavailable with *lpc.*

Warning: no daemon present

The *lpd* process overseeing the spooling queue, as indicated in the "lock" file in that directory, does not exist. This normally occurs only when the daemon has unexpectedly died. The error log file for the printer should be checked for a diagnostic from the deceased process. To restart an *lpd,* use

        % lpc restart *printer*

7.3. LPRM

lprm: *printer* : **cannot restart printer daemon**

This case is the same as when *lpr* prints that the daemon cannot be started.

7.4. LPD

The *lpd* program can write many different messages to the error log file (the file specified in the **lf** entry in *printcap*). Most of these messages are about files which can not be opened and usually indicate the *printcap* file or the protection modes of the files are not correct. Files may also be inaccessible if people manually manipulate the line printer system (i.e. they bypass the *lpr* program).

In addition to messages generated by *lpd,* any of the filters that *lpd* spawns may also log messages to this file.

7.5. LPC

could't start printer

This case is the same as when *lpr* reports that the daemon cannot be started.

cannot examine spool directory

Error messages beginning with "cannot ..." are usually due to incorrect ownership and/or protection mode of the lock file, spooling directory or the *lpc* program.

A Dial-Up Network of UNIX™ Systems

D. A. Nowitz

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

A network of over eighty UNIX† computer systems has been established using the telephone system as its primary communication medium. The network was designed to meet the growing demands for software distribution and exchange. Some advantages of our design are:

- The startup cost is low. A system needs only a dial-up port, but systems with automatic calling units have much more flexibility.

- No operating system changes are required to install or use the system.

- The communication is basically over dial-up lines, however, hardwired communication lines can be used to increase speed.

- The command for sending/receiving files is simple to use.

Keywords: networks, communications, software distribution, software maintenance

August 18, 1978

---

† UNIX is a Trademark of Bell Laboratories.

A Dial-Up Network of UNIX™ Systems

D. A. Nowitz

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

1. Purpose

The widespread use of the UNIX† system[1] within Bell Laboratories has produced problems of software distribution and maintenance. A conventional mechanism was set up to distribute the operating system and associated programs from a central site to the various users. However this mechanism alone does not meet all software distribution needs. Remote sites generate much software and must transmit it to other sites. Some UNIX systems are themselves central sites for redistribution of a particular specialized utility, such as the Switching Control Center System. Other sites have particular, often long-distance needs for software exchange; switching research, for example, is carried on in New Jersey, Illinois, Ohio, and Colorado. In addition, general purpose utility programs are written at all UNIX system sites. The UNIX system is modified and enhanced by many people in many places and it would be very constricting to deliver new software in a one-way stream without any alternative for the user sites to respond with changes of their own.

Straightforward software distribution is only part of the problem. A large project may exceed the capacity of a single computer and several machines may be used by the one group of people. It then becomes necessary for them to pass messages, data and other information back an forth between computers.

Several groups with similar problems, both inside and outside of Bell Laboratories, have constructed networks built of hardwired connections only.[2,3] Our network, however, uses both dial-up and hardwired connections so that service can be provided to as many sites as possible.

2. Design Goals

Although some of our machines are connected directly, others can only communicate over low-speed dial-up lines. Since the dial-up lines are often unavailable and file transfers may take considerable time, we spool all work and transmit in the background. We also had to adapt to a community of systems which are independently operated and resistant to suggestions that they should all buy particular hardware or install particular operating system modifications. Therefore, we make minimal demands on the local sites in the network. Our implementation requires no operating system changes; in fact, the transfer programs look like any other user entering the system through the normal dial-up login ports, and obeying all local protection rules.

We distinguish "active" and "passive" systems on the network. Active systems have an automatic calling unit or a hardwired line to another system, and can initiate a connection. Passive systems do not have the hardware to initiate a connection. However, an active system can be assigned the job of calling passive systems and executing work found there; this makes a passive system the functional equivalent of an active system, except for an additional delay while it waits to be polled. Also, people frequently log into active systems and request copying from one passive system to another. This requires two telephone calls, but even so, it is faster

---

†UNIX is a Trademark of Bell Laboratories.

than mailing tapes.

Where convenient, we use hardwired communication lines. These permit much faster transmission and multiplexing of the communications link. Dial-up connections are made at either 300 or 1200 baud; hardwired connections are asynchronous up to 9600 baud and might run even faster on special-purpose communications hardware.[4,5] Thus, systems typically join our network first as passive systems and when they find the service more important, they acquire automatic calling units and become active systems; eventually, they may install high-speed links to particular machines with which they handle a great deal of traffic. At no point, however, must users change their programs or procedures.

The basic operation of the network is very simple. Each participating system has a spool directory, in which work to be done (files to be moved, or commands to be executed remotely) is stored. A standard program, *uucico*, performs all transfers. This program starts by identifying a particular communication channel to a remote system with which it will hold a conversation. *Uucico* then selects a device and establishes the connection, logs onto the remote machine and starts the *uucico* program on the remote machine. Once two of these programs are connected, they first agree on a line protocol, and then start exchanging work. Each program in turn, beginning with the calling (active system) program, transmits everything it needs, and then asks the other what it wants done. Eventually neither has any more work, and both exit.

In this way, all services are available from all sites; passive sites, however, must wait until called. A variety of protocols may be used; this conforms to the real, non-standard world. As long as the caller and called programs have a protocol in common, they can communicate. Furthermore, each caller knows the hours when each destination system should be called. If a destination is unavailable, the data intended for it remain in the spool directory until the destination machine can be reached.

The implementation of this Bell Laboratories network between independent sites, all of which store proprietary programs and data, illustratives the pervasive need for security and administrative controls over file access. Each site, in configuring its programs and system files, limits and monitors transmission. In order to access a file a user needs access permission for the machine that contains the file and access permission for the file itself. This is achieved by first requiring the user to use his password to log into his local machine and then his local machine logs into the remote machine whose files are to be accessed. In addition, records are kept identifying all files that are moved into and out of the local system, and how the requestor of such accesses identified himself. Some sites may arrange to permit users only to call up and request work to be done; the calling users are then called back before the work is actually done. It is then possible to verify that the request is legitimate from the standpoint of the target system, as well as the originating system. Furthermore, because of the call-back, no site can masquerade as another even if it knows all the necessary passwords.

Each machine can optionally maintain a sequence count for conversations with other machines and require a verification of the count at the start of each conversation. Thus, even if call back is not in use, a successful masquerade requires the calling party to present the correct sequence number. A would-be impersonator must not just steal the correct phone number, user name, and password, but also the sequence count, and must call in sufficiently promptly to precede the next legitimate request from either side. Even a successful masquerade will be detected on the next correct conversation.

3. Processing

The user has two commands which set up communications, *uucp* to set up file copying, and *uux* to set up command execution where some of the required resources (system and/or files) are not on the local machine. Each of these commands will put work and data files into the spool directory for execution by *uucp* daemons. Figure 1 shows the major blocks of the file transfer process.

File Copy

The *uucico* program is used to perform all communications between the two systems. It performs the following functions:

- Scan the spool directory for work.
- Place a call to a remote system.
- Negotiate a line protocol to be used.
- Start program *uucico* on the remote system.
- Execute all requests from both systems.
- Log work requests and work completions.

Uucico may be started in several ways;

   a)   by a system daemon,
   b)   by one of the *uucp* or *uux* programs,
   c)   by a remote system.

Scan For Work

The file names in the spool directory are constructed to allow the daemon programs *(uucico, uuxqt)* to determine the files they should look at, the remote machines they should call and the order in which the files for a particular remote machine should be processed.

Call Remote System

The call is made using information from several files which reside in the uucp program directory. At the start of the call process, a lock is set on the system being called so that another call will not be attempted at the same time.

The system name is found in a "systems" file. The information contained for each system is:

   [1]   system name,
   [2]   times to call the system (days-of-week and times-of-day),
   [3]   device or device type to be used for call,
   [4]   line speed,
   [5]   phone number,
   [6]   login information (multiple fields).

The time field is checked against the present time to see if the call should be made. The *phone number* may contain abbreviations (e.g. "nyc", "boston") which get translated into dial sequences using a "dial-codes" file. This permits the same "phone number" to be stored at every site, despite local variations in telephone services and dialing conventions.

A "devices" file is scanned using fields [3] and [4] from the "systems" file to find an available device for the connection. The program will try all devices which satisfy [3] and [4] until a connection is made, or no more devices can be tried. If a non-multiplexable device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the connection is complete, the *login information* is used to log into the remote system. Then a command is sent to the remote system to start the *uucico* program. The conversation between the two *uucico* programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins.

Line Protocol Selection

The remote system sends a message

P *proto-list*

where *proto-list* is a string of characters, each representing a line protocol. The calling program checks the proto-list for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

U*code*

where code is either a one character protocol letter or a *N* which means there is no common protocol.

Greg Chesson designed and implemented the standard line protocol used by the uucp transmission program. Other protocols may be added by individual installations.

Work Processing

During processing, one program is the *MASTER* and the other is *SLAVE*. Initially, the calling program is the *MASTER*. These roles may switch one or more times during the conversation.

There are four messages used during the work processing, each specified by the first character of the message. They are

S   send a file,
R   receive a file,
C   copy complete,
H   hangup.

The *MASTER* will send *R* or *S* messages until all work from the spool directory is complete, at which point an *H* message will be sent. The *SLAVE* will reply with *SY, SN, RY, RN, HY, HN*, corresponding to *yes* or *no* for each request.

The send and receive replies are based on permission to access the requested file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message *CY* will be sent if the UNIX *cp* command, used to copy from the spool directory, is successful. Otherwise, a *CN* message is sent. The requests and results are logged on both systems, and, if requested, mail is sent to the user reporting completion (or the user can request status information from the log program at any time).

The hangup response is determined by the *SLAVE* program by a work scan of the spool directory. If work for the remote system exists in the *SLAVE's* spool directory, a *HN* message is sent and the programs switch roles. If no work exists, an *HY* response is sent.

A sample conversation is shown in Figure 2.

Conversation Termination

When a *HY* message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final "OO" message to the other.

4. Present Uses

One application of this software is remote mail. Normally, a UNIX system user writes "mail dan" to send mail to user "dan". By writing "mail usg!dan" the mail is sent to user "dan" on system "usg".

The primary uses of our network to date have been in software maintenance. Relatively few of the bytes passed between systems are intended for people to read. Instead, new programs (or new versions of programs) are sent to users, and potential bugs are returned to authors. Aaron Cohen has implemented a "stockroom" which allows remote users to call in

and request software. He keeps a "stock list" of available programs, and new bug fixes and utilities are added regularly. In this way, users can always obtain the latest version of anything without bothering the authors of the programs. Although the stock list is maintained on a particular system, the items in the stockroom may be warehoused in many places; typically each program is distributed from the home site of its author. Where necessary, uucp does remote-to-remote copies.

We also routinely retrieve test cases from other systems to determine whether errors on remote systems are caused by local misconfigurations or old versions of software, or whether they are bugs that must be fixed at the home site. This helps identify errors rapidly. For one set of test programs maintained by us, over 70% of the bugs reported from remote sites were due to old software, and were fixed merely by distributing the current version.

Another application of the network for software maintenance is to compare files on two different machines. A very useful utility on one machine has been Doug McIlroy's "diff" program which compares two text files and indicates the differences, line by line, between them.[6] Only lines which are not identical are printed. Similarly, the program "uudiff" compares files (or directories) on two machines. One of these directories may be on a passive system. The "uudiff" program is set up to work similarly to the inter-system mail, but it is slightly more complicated.

To avoid moving large numbers of usually identical files, *uudiff* computes file checksums on each side, and only moves files that are different for detailed comparison. For large files, this process can be iterated; checksums can be computed for each line, and only those lines that are different actually moved.

The "uux" command has been useful for providing remote output. There are some machines which do not have hard-copy devices, but which are connected over 9600 baud communication lines to machines with printers. The *uux* command allows the formatting of the printout on the local machine and printing on the remote machine using standard UNIX command programs.

5. Performance

Throughput, of course, is primarily dependent on transmission speed. The table below shows the real throughput of characters on communication links of different speeds. These numbers represent actual data transferred; they do not include bytes used by the line protocol for data validation such as checksums and messages. At the higher speeds, contention for the processors on both ends prevents the network from driving the line full speed. The range of speeds represents the difference between light and heavy loads on the two systems. If desired, operating system modifications can be installed that permit full use of even very fast links.

Nominal speed	Characters/sec.
300 baud	27
1200 baud	100-110
9600 baud	200-850

In addition to the transfer time, there is some overhead for making the connection and logging in ranging from 15 seconds to 1 minute. Even at 300 baud, however, a typical 5,000 byte source program can be transferred in four minutes instead of the 2 days that might be required to mail a tape.

Traffic between systems is variable. Between two closely related systems, we observed 20 files moved and 5 remote commands executed in a typical day. A more normal traffic out of a single system would be around a dozen files per day.

The total number of sites at present in the main network is 82, which includes most of the Bell Laboratories full-size machines which run the UNIX operating system. Geographically, the machines range from Andover, Massachusetts to Denver, Colorado.

Uucp has also been used to set up another network which connects a group of systems in operational sites with the home site. The two networks touch at one Bell Labs computer.

6. Further Goals

Eventually, we would like to develop a full system of remote software maintenance. Conventional maintenance (a support group which mails tapes) has many well-known disadvantages.[7] There are distribution errors and delays, resulting in old software running at remote sites and old bugs continually reappearing. These difficulties are aggravated when there are 100 different small systems, instead of a few large ones.

The availability of file transfer on a network of compatible operating systems makes it possible just to send programs directly to the end user who wants them. This avoids the bottleneck of negotiation and packaging in the central support group. The "stockroom" serves this function for new utilities and fixes to old utilities. However, it is still likely that distributions will not be sent and installed as often as needed. Users are justifiably suspicious of the "latest version" that has just arrived; all too often it features the "latest bug." What is needed is to address both problems simultaneously:

1. Send distributions whenever programs change.

2. Have sufficient quality control so that users will install them.

To do this, we recommend systematic regression testing both on the distributing and receiving systems. Acceptance testing on the receiving systems can be automated and permits the local system to ensure that its essential work can continue despite the constant installation of changes sent from elsewhere. The work of writing the test sequences should be recovered in lower counseling and distribution costs.

Some slow-speed network services are also being implemented. We now have inter-system "mail" and "diff," plus the many implied commands represented by "uux." However, we still need inter-system "write" (real-time inter-user communication) and "who" (list of people logged in on different systems). A slow-speed network of this sort may be very useful for speeding up counseling and education, even if not fast enough for the distributed data base applications that attract many users to networks. Effective use of remote execution over slow-speed lines, however, must await the general installation of multiplexable channels so that long file transfers do not lock out short inquiries.

7. Lessons

The following is a summary of the lessons we learned in building these programs.

1. By starting your network in a way that requires no hardware or major operating system changes, you can get going quickly.

2. Support will follow use. Since the network existed and was being used, system maintainers were easily persuaded to help keep it operating, including purchasing additional hardware to speed traffic.

3. Make the network commands look like local commands. Our users have a resistance to learning anything new: all the inter-system commands look very similar to standard UNIX system commands so that little training cost is involved.

4. An initial error was not coordinating enough with existing communications projects: thus, the first version of this network was restricted to dial-up, since it did not support the various hardware links between systems. This has been fixed in the current system.

References

1. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.* 57(6) pp. 1905-1929 (1978).

2. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "UNIX Time-Sharing System: The Programmer's Workbench," *Bell Sys. Tech. J.* 57(6) pp. 2177-2200 (1978).

3. G. L. Chesson, "The Network UNIX System," *Operating Systems Review* 9(5) pp. 60-66 (1975). Also in *Proc. 5th Symp. on Operating Systems Principles.*

4. A. G. Fraser, "Spider — An Experimental Data Communications System," *Proc. IEEE Conf. on Communications,* p. 21F (June 1974). IEEE Cat. No. 74CH0859-9-CSCB.

5. A. G. Fraser, "A Virtual Channel Network," *Datamation,* pp. 51-56 (February 1975).

6. J. W. Hunt and M. D. McIlroy, "An Algorithm for Differential File Comparison," Comp. Sci. Tech. Rep. No. 41, Bell Laboratories, Murray Hill, New Jersey (June 1976).

7. F. P. Brooks, Jr., *The Mythical Man-Month.* Addison-Wesley, Reading, Mass. (1975).

Uucp Implementation Description

D. A. Nowitz

ABSTRACT

Uucp is a series of programs designed to permit communication between UNIX systems using either dial-up or hardwired communication lines. This document gives a detailed implementation description of the current (second) implementation of uucp.

This document is for use by an administrator/installer of the system. It is not meant as a user's guide.

October 31, 1978

Uucp Implementation Description

D. A. Nowitz

Introduction

Uucp is a series of programs designed to permit communication between UNIX† systems using either dial-up or hardwired communication lines. It is used for file transfers and remote command execution. The first version of the system was designed and implemented by M. E. Lesk.[1] This paper describes the current (second) implementation of the system.

Uucp is a batch type operation. Files are created in a spool directory for processing by the uucp demons. There are three types of files used for the execution of work. *Data files* contain data for transfer to remote systems. *Work files* contain directions for file transfers between systems. *Execution files* are directions for UNIX command executions which involve the resources of one or more systems.

The uucp system consists of four primary and two secondary programs. The primary programs are:

uucp   This program creates work and gathers data files in the spool directory for the transmission of files.

uux   This program creates work files, execute files and gathers data files for the remote execution of UNIX commands.

uucico This program executes the work files for data transmission.

uuxqt This program executes the execution files for UNIX command execution.

The secondary programs are:

uulog This program updates the log file with new entries and reports on the status of uucp requests.

uuclean This program removes old files from the spool directory.

The remainder of this paper will describe the operation of each program, the installation of the system, the security aspects of the system, the files required for execution, and the administration of the system.

1. Uucp - UNIX to UNIX File Copy

The *uucp* command is the user's primary interface with the system. The *uucp* command was designed to look like *cp* to the user. The syntax is

uucp | option | ... source ... destination

where the source and destination may contain the prefix *system-name!* which indicates the system on which the file or files reside or where they will be copied.

The options interpreted by *uucp* are:

−d   Make directories when necessary for copying the file.

---

†UNIX is a Trademark of Bell Laboratories.
1 M. E. Lesk and A. S. Cohen, UNIX Software Distribution by Communication Link, private communication.

−c Don't copy source files to the spool directory, but use the specified source when the actual transfer takes place.

−g *letter* Put *letter* in as the grade in the name of the work file. (This can be used to change the order of work for a particular machine.)

−m Send mail on completion of the work.

The following options are used primarily for debugging:

−r Queue the job but do not start *uucico* program.

−s*dir* Use directory *dir* for the spool directory.

−x *num* *Num* is the level of debugging output desired.

The destination may be a directory name, in which case the file name is taken from the last part of the source's name. The source name may contain special shell characters such as " *?*[]". If a source argument has a *system-name!* prefix for a remote system, the file name expansion will be done on the remote system.

The command

  uucp *.c usg!/usr/dan

will set up the transfer of all files whose names end with ".c" to the "/usr/dan" directory on the"usg" machine.

The source and/or destination names may also contain a ~*user* prefix. This translates to the login directory on the specified system. For names with partial path-names, the current directory is prepended to the file name. File names with ../ are not permitted.

The command

  uucp usg!~dan/*.h ~dan

will set up the transfer of files whose names end with ".h" in dan's login directory on system "usg" to dan's local login directory.

For each source file, the program will check the source and destination file-names and the system-part of each to classify the work into one of five types:

[1] Copy source to destination on local system.

[2] Receive files from other systems.

[3] Send files to a remote systems.

[4] Send files from remote systems to another remote system.

[5] Receive files from remote systems when the source contains special shell characters as mentioned above.

After the work has been set up in the spool directory, the *uucico* program is started to try to contact the other machine to execute the work (unless the −r option was specified).

Type 1

A *cp* command is used to do the work. The −d and the −m options are not honored in this case.

Type 2

A one line *work file* is created for each file requested and put in the spool directory with the following fields, each separated by a blank. (All *work files* and *execute files* use a blank as the field separator.)

  [1] R

[2] The full path-name of the source or a ¯user/path-name. The ¯user part will be expanded on the remote system.

[3] The full path-name of the destination file. If the ¯user notation is used, it will be immediately expanded to be the login directory for the user.

[4] The user's login name.

[5] A "−" followed by an option list. (Only the −m and −d options will appear in this list.)

Type 3

For each source file, a *work file* is created and the source file is copied into a *data file* in the spool directory. (A "−c" option on the *uucp* command will prevent the *data file* from being made.) In this case, the file will be transmitted from the indicated source.) The fields of each entry are given below.

[1] S

[2] The full-path name of the source file.

[3] The full-path name of the destination or ¯user/file-name.

[4] The user's login name.

[5] A "−" followed by an option list.

[6] The name of the *data file* in the spool directory.

[7] The file mode bits of the source file in octal print format (e.g. 0666).

Type 4 and Type 5

Uucp generates a *uucp* command and sends it to the remote machine; the remote *uucico* executes the *uucp* command.

2. Uux - UNIX To UNIX Execution

The *uux* command is used to set up the execution of a UNIX command where the execution machine and/or some of the files are remote. The syntax of the uux command is

    uux [−] [option] ... command-string

where the command-string is made up of one or more arguments. All special shell characters such as "< >|" must be quoted either by quoting the entire command-string or quoting the character as a separate argument. Within the command-string, the command and file names may contain a *system-name!* prefix. All arguments which do not contain a "!" will not be treated as files. (They will not be copied to the execution machine.) The "−" is used to indicate that the standard input for *command-string* should be inherited from the standard input of the *uux* command. The options, essentially for debugging, are:

    −r   Don't start *uucico* or *uuxqt* after queuing the job;

    −x*num*   Num is the level of debugging output desired.

The command

    pr abc | uux − usg!lpr

will set up the output of "pr abc" as standard input to an lpr command to be executed on system "usg".

Uux generates an *execute file* which contains the names of the files required for execution (including standard input), the user's login name, the destination of the standard output, and the command to be executed. This file is either put in the spool directory for local execution or sent to the remote system using a generated send command (type 3 above).

For required files which are not on the execution machine, *uux* will generate receive command files (type 2 above). These command-files will be put on the execution machine and executed

by the *uucico* program. (This will work only if the local system has permission to put files in the remote spool directory as controlled by the remote *USERFILE.*)

The *execute file* will be processed by the *uuxqt* program on the execution machine. It is made up of several lines, each of which contains an identification character and one or more arguments. The order of the lines in the file is not relevant and some of the lines may not be present. Each line is described below.

User Line

   U user system

where the *user* and *system* are the requester's login name and system.

Required File Line

   F file-name real-name

where the *file-name* is the generated name of a file for the execute machine and *real-name* is the last part of the actual file name (contains no path information). Zero or more of these lines may be present in the *execute file*. The *uuxqt* program will check for the existence of all required files before the command is executed.

Standard Input Line

   I file-name

The standard input is either specified by a "<" in the command-string or inherited from the standard input of the *uux* command if the "−" option is used. If a standard input is not specified, "/dev/null" is used.

Standard Output Line

   O file-name system-name

The standard output is specified by a ">" within the command-string. If a standard output is not specified, "/dev/null" is used. (Note − the use of ">>" is not implemented.)

Command Line

   C command [arguments] ...

The arguments are those specified in the command-string. The standard input and standard output will not appear on this line. All *required files* will be moved to the execution directory (a subdirectory of the spool directory) and the UNIX command is executed using the Shell specified in the *uucp.h* header file. In addition, a shell "PATH" statement is prepended to the command line as specified in the *uuxqt* program.

After execution, the standard output is copied or set up to be sent to the proper place.

3. Uucico - Copy In, Copy Out

The *uucico* program will perform the following major functions:

   - Scan the spool directory for work.
   - Place a call to a remote system.
   - Negotiate a line protocol to be used.
   - Execute all requests from both systems.
   - Log work requests and work completions.

Uucico may be started in several ways;

a) by a system daemon,

b) by one of the *uucp, uux, uuxqt* or *uucico* programs,

c) directly by the user (this is usually for testing),

d) by a remote system. (The uucico program should be specified as the "shell" field in the "/etc/passwd" file for the "uucp" logins.)

When started by method a, b or c, the program is considered to be in *MASTER* mode. In this mode, a connection will be made to a remote system. If started by a remote system (method d), the program is considered to be in *SLAVE* mode.

The *MASTER* mode will operate in one of two ways. If no system name is specified (−s option not specified) the program will scan the spool directory for systems to call. If a system name is specified, that system will be called, and work will only be done for that system.

The *uucico* program is generally started by another program. There are several options used for execution:

−r1   Start the program in *MASTER* mode. This is used when *uucico* is started by a program or "cron" shell.

−s*sys*   Do work only for system *sys*. If −s is specified, a call to the specified system will be made even if there is no work for system *sys* in the spool directory. This is useful for polling systems which do not have the hardware to initiate a connection.

The following options are used primarily for debugging:

−d*dir*   Use directory *dir* for the spool directory.

−x*num*   *Num* is the level of debugging output desired.

The next part of this section will describe the major steps within the *uucico* program.

Scan For Work

The names of the work related files in the spool directory have format

    type . system-name grade number

where:

Type is an upper case letter, (*C* - copy command file, *D* - data file, *X* - execute file);

System-name is the remote system;

Grade is a character;

Number is a four digit, padded sequence number.

The file

    C.res45n0031

would be a *work file* for a file transfer between the local machine and the "res45" machine.

The scan for work is done by looking through the spool directory for *work files* (files with prefix "C."). A list is made of all systems to be called. *Uucico* will then call each system and process all *work files*.

Call Remote System

The call is made using information from several files which reside in the uucp program directory. At the start of the call process, a lock is set to forbid multiple conversations between the same two systems.

The system name is found in the *L.sys* file. The information contained for each system is:

[1] system name,

[2] times to call the system (days-of-week and times-of-day),

[3] device or device type to be used for call,

[4] line speed,

[5] phone number if field [3] is *ACU* or the device name (same as field [3]) if not *ACU*,

[6] login information (multiple fields),

The time field is checked against the present time to see if the call should be made.

The *phone number* may contain abbreviations (e.g. mh, py, boston) which get translated into dial sequences using the *L-dialcodes* file.

The *L-devices* file is scanned using fields [3] and [4] from the *L.sys* file to find an available device for the call. The program will try all devices which satisfy [3] and [4] until the call is made, or no more devices can be tried. If a device is successfully opened, a lock file is created so that another copy of *uucico* will not try to use it. If the call is complete, the *login information* (field [6] of *L.sys*) is used to login.

The conversation between the two *uucico* programs begins with a handshake started by the called, *SLAVE*, system. The *SLAVE* sends a message to let the *MASTER* know it is ready to receive the system identification and conversation sequence number. The response from the *MASTER* is verified by the *SLAVE* and if acceptable, protocol selection begins. The *SLAVE* can also reply with a "call-back required" message in which case, the current conversation is terminated.

Line Protocol Selection

The remote system sends a message

P*proto-list*

where proto-list is a string of characters, each representing a line protocol.

The calling program checks the proto-list for a letter corresponding to an available line protocol and returns a *use-protocol* message. The *use-protocol* message is

U*code*

where code is either a one character protocol letter or *N* which means there is no common protocol.

Work Processing

The initial roles (*MASTER* or *SLAVE*) for the work processing are the mode in which each program starts. (The *MASTER* has been specified by the "−r1" uucico option.) The *MASTER* program does a work search similar to the one used in the "Scan For Work" section.

There are five messages used during the work processing, each specified by the first character of the message. They are;

S send a file,

R receive a file,

C copy complete,

X execute a *uucp* command,

H hangup.

The *MASTER* will send *R, S* or *X* messages until all work from the spool directory is complete, at which point an *H* message will be sent. The *SLAVE* will reply with *SY, SN, RY, RN, HY, HN, XY, XN, corresponding to yes or no for each request.*

The send and receive replies are based on permission to access the requested file/directory using the *USERFILE* and read/write permissions of the file/directory. After each file is copied into the spool directory of the receiving system, a copy-complete message is sent by the receiver of the file. The message *CY* will be sent if the file has successfully been moved from the temporary spool file to the actual destination. Otherwise, a *CN* message is sent. (In the case of *CN*, the transferred file will be in the spool directory with a name beginning with "TM'.) The requests and results are logged on both systems.

The hangup response is determined by the *SLAVE* program by a work scan of the spool directory. If work for the remote system exists in the *SLAVE's* spool directory, an *HN* message is sent and the programs switch roles. If no work exists, an *HY* response is sent.

Conversation Termination

When a *HY* message is received by the *MASTER* it is echoed back to the *SLAVE* and the protocols are turned off. Each program sends a final "OO" message to the other. The original *SLAVE* program will clean up and terminate. The *MASTER* will proceed to call other systems and process work as long as possible or terminate if a −s option was specified.

4. Uuxqt - Uucp Command Execution

The *uuxqt* program is used to execute *execute files* generated by *uux*. The *uuxqt* program may be started by either the *uucico* or *uux* programs. The program scans the spool directory for *execute files* (prefix "X."). Each one is checked to see if all the required files are available and if so, the command line or send line is executed.

The *execute file* is described in the "Uux" section above.

Command Execution

The execution is accomplished by executing a *sh* −c of the command line after appropriate standard input and standard output have been opened. If a standard output is specified, the program will create a send command or copy the output file as appropriate.

5. Uulog - Uucp Log Inquiry

The *uucp* programs create individual log files for each program invocation. Periodically, *uulog* may be executed to prepend these files to the system logfile. This method of logging was chosen to minimize file locking of the logfile during program execution.

The *uulog* program merges the individual log files and outputs specified log entries. The output request is specified by the use of the following options:

−s*sys*   Print entries where *sys* is the remote system name;

−u*user*   Print entries for user *user*.

The intersection of lines satisfying the two options is output. A null *sys* or *user* means all system names or users respectively.

6. Uuclean - Uucp Spool Directory Cleanup

This program is typically started by the daemon, once a day. Its function is to remove files from the spool directory which are more than 3 days old. These are usually files for work which can not be completed.

The options available are:

−d*dir*   The directory to be scanned is *dir*.

−m   Send mail to the owner of each file being removed. (Note that most files put into the spool directory will be owned by the owner of the uucp programs since the setuid bit will be set on these programs. The mail will therefore most often go to the owner of the uucp programs.)

- n *hours*   Change the aging time from 72 hours to *hours* hours.
- p *pre*   Examine files with prefix *pre* for deletion. (Up to 10 file prefixes may be specified.)
- x *num*   This is the level of debugging output desired.

7. Security

The uucp system, left unrestricted, will let any outside user execute any commands and copy in/out any file which is readable/writable by the uucp login user. It is up to the individual sites to be aware of this and apply the protections that they feel are necessary.

There are several security features available aside from the normal file mode protections. These must be set up by the installer of the *uucp* system.

- The login for uucp does not get a standard shell. Instead, the *uucico* program is started. Therefore, the only work that can be done is through *uucico*.
- A path check is done on file names that are to be sent or received. The *USERFILE* supplies the information for these checks. The *USERFILE* can also be set up to require call-back for certain login-ids. (See the "Files required for execution" section for the file description.)
- A conversation sequence count can be set up so that the called system can be more confident that the caller is who he says he is.
- The *uuxqt* program comes with a list of commands that it will execute. A "PATH" shell statement is prepended to the command line as specifed in the *uuxqt* program. The installer may modify the list or remove the restrictions as desired.
- The *L.sys* file should be owned by uucp and have mode 0400 to protect the phone numbers and login information for remote sites. (Programs uucp, uucico, uux, uuxqt should be also owned by uucp and have the setuid bit set.)

8. Uucp Installation

There are several source modifications that may be required before the system programs are compiled. These relate to the directories used during compilation, the directories used during execution, and the local *uucp system-name*.

The four directories are:

    lib   (/usr/src/cmd/uucp) This directory contains the source files for generating the *uucp* system.

    program   (/usr/lib/uucp) This is the directory used for the executable system programs and the system files.

    spool   (/usr/spool/uucp) This is the spool directory used during *uucp* execution.

    xqtdir   (/usr/spool/uucp/.XQTDIR) This directory is used during execution of *execute files*.

The names given in parentheses above are the default values for the directories. The italicized named *lib, program, xqtdir,* and *spool* will be used in the following text to represent the appropriate directory names.

There are two files which may require modification, the *makefile* file and the *uucp.h* file. The following paragraphs describe the modifications. The modes of *spool* and *xqtdir* should be made "0777".

Uucp.h modification

Change the *program* and the *spool* names from the default values to the directory names to be used on the local system using global edit commands.

Change the *define* value for *MYNAME* to be the local *uucp* system-name.

makefile modification

There are several *make* variable definitions which may need modification.

INSDIR   This is the *program* directory (e.g. INSDIR=/usr/lib/uucp). This parameter is used if "make cp" is used after the programs are compiled.

IOCTL   This is required to be set if an appropriate *ioctl* interface subroutine does not exist in the standard "C" library; the statement "IOCTL=ioctl.o" is required in this case.

PKON   The statement "PKON=pkon.o" is required if the packet driver is not in the kernel.

Compile the system The command

      make

will compile the entire system. The command

      make cp

will copy the commands to the to the appropriate directories.

The programs *uucp*, *uux*, and *uulog* should be put in "/usr/bin". The programs *uuxqt*, *uucico*, and *uuclean* should be put in the *program* directory.

Files required for execution

There are four files which are required for execution, all of which should reside in the *program* directory. The field separator for all files is a space unless otherwise specified.

L-devices

This file contains entries for the call-unit devices and hardwired connections which are to be used by *uucp*. The special device files are assumed to be in the */dev* directory. The format for each entry is

      line call-unit speed

where;

line   is the device for the line (e.g. cul0),

call-unit   is the automatic call unit associated with *line* (e.g. cua0), (Hardwired lines have a number "0" in this field.),

speed   is the line speed.

The line

      cul0 cua0 300

would be set up for a system which had device cul0 wired to a call-unit cua0 for use at 300 baud.

L-dialcodes

This file contains entries with location abbreviations used in the *L.sys* file (e.g. py, mh, boston). The entry format is

          abb dial-seq

where;

     abb   is the abbreviation,

     dial-seq   is the dial sequence to call that location.

The line

          py 165 —

would be set up so that entry py7777 would send 165 — 7777 to the dial-unit.

LOGIN/SYSTEM NAMES

It is assumed that the *login name* used by a remote computer to call into a local computer is not the same as the login name of a normal user of that local machine. However, several remote computers may employ the same login name.

Each computer is given a unique *system name* which is transmitted at the start of each call. This name identifies the calling machine to the called machine.

USERFILE

This file contains user accessibility information. It specifies four types of constraint;

     [1]   which files can be accessed by a normal user of the local machine,

     [2]   which files can be accessed from a remote computer,

     [3]   which login name is used by a particular remote computer,

     [4]   whether a remote computer should be called back in order to confirm its identity.

Each line in the file has the following format

     login,sys [c] path-name [path-name] ...

where;

     login   is the login name for a user or the remote computer,

     sys   is the system name for a remote computer,

     c   is the optional *call-back required* flag,

     path-name   is a path-name prefix that is acceptable for *user*.

The constraints are implemented as follows.

     [1]   When the program is obeying a command stored on the local machine, *MASTER* mode, the path-names allowed are those given for the first line in the *USERFILE* that has a login name that matches the login name of the user who entered the command. If no such line is found, the first line with a *null* login name is used.

     [2]   When the program is responding to a command from a remote machine, *SLAVE* mode, the path-names allowed are those given for the first line in the file that has the system name that matches the system name of the remote machine. If no such line is found, the first one with a *null* system name is used.

     [3]   When a remote computer logs in, the login name that it uses must appear in the *USERFILE*. There may be several lines with the same login name but one of them must either have the name of the remote system or must contain a *null* system name.

     [4]   If the line matched in ([3]) contains a "c", the remote machine is called back before any transactions take place.

The line

        u,m /usr/xyz

allows machine *m* to login with name *u* and request the transfer of files whose names start with "/usr/xyz".

The line

        dan, /usr/dan

allows the ordinary user *dan* to issue commands for files whose name starts with "/usr/dan".

The lines

        u,m /usr/xyz /usr/spool
        u, /usr/spool

allows any remote machine to login with name *u*, but if its system name is not *m*, it can only ask to transfer files whose names start with "/usr/spool".

The lines

        root, /
            /usr

allows any user to transfer files beginning with "/usr" but the user with login *root* can transfer any file.

L.sys

Each entry in this file represents one system which can be called by the local uucp programs. The fields are described below.

system name

The name of the remote system.

time

This is a string which indicates the days-of-week and times-of-day when the system should be called (e.g. MoTuTh0800−1730).

The day portion may be a list containing some of

   Su Mo Tu We Th Fr Sa

or it may be *Wk* for any week-day or *Any* for any day.

The time should be a range of times (e.g. 0800−1230). If no time portion is specified, any time of day is assumed to be ok for the call.

device

This is either *ACU* or the hardwired device to be used for the call. For the hardwired case, the last part of the special file name is used (e.g. tty0).

speed

This is the line speed for the call (e.g. 300).

phone

The phone number is made up of an optional alphabetic abbreviation and a numeric part. The abbreviation is one which appears in the *L-dialcodes* file (e.g. mh5900, boston995−9980).

For the hardwired devices, this field contains the same string as used for the *device* field.

login

The login information is given as a series of fields and subfields in the format

    expect send [expect send] ...

where; *expect* is the string expected to be read and *send* is the string to be sent when the *expect* string is received.

The expect field may be made up of subfields of the form

    expect[−send−expect]...

where the *send* is sent if the prior *expect* is not successfully read and the *expect* following the *send* is the next expected string.

There are two special names available to be sent during the login sequence. The string *EOT* will send an EOT character and the string *BREAK* will try to send a BREAK character. (The *BREAK* character is simulated using line speed changes and null characters and may not work on all devices and/or systems.)

A typical entry in the L.sys file would be

    sys Any ACU 300 mh7654 login uucp ssword: word

The expect algorithm looks at the last part of the string as illustrated in the password field.

9. Administration

This section indicates some events and files which must be administered for the *uucp* system. Some administration can be accomplished by *shell files* which can be initiated by *crontab* entries. Others will require manual intervention. Some sample *shell files* are given toward the end of this section.

SQFILE − sequence check file

This file is set up in the *program* directory and contains an entry for each remote system with which you agree to perform conversation sequence checks. The initial entry is just the system name of the remote system. The first conversation will add two items to the line, the conversation count, and the date/time of the most resent conversation. These items will be updated with each conversation. If a sequence check fails, the entry will have to be adjusted.

TM − temporary data files

These files are created in the *spool* directory while files are being copied from a remote machine. Their names have the form

    TM.pid.ddd

where *pid* is a process-id and *ddd* is a sequential three digit number starting at zero for each invocation of *uucico* and incremented for each file received.

After the entire remote file is received, the *TM* file is moved/copied to the requested destination. If processing is abnormally terminated or the move/copy fails, the file will remain in the spool directory.

The leftover files should be periodically removed; the *uuclean* program is useful in this regard. The command

    uuclean −pTM

will remove all *TM* files older than three days.

LOG — log entry files

During execution of programs, individual *LOG* files are created in the *spool* directory with information about queued requests, calls to remote systems, execution of *uux* commands and file copy results. These files should be combined into the *LOGFILE* by using the *uulog* program. This program will put the new *LOG* files at the beginning of the existing *LOGFILE*. The command

        uulog

will accomplish the merge. Options are available to print some or all the log entries after the files are merged. The *LOGFILE* should be removed periodically since it is copied each time new LOG entries are put into the file.

The *LOG* files are created initially with mode 0222. If the program which creates the file terminates normally, it changes the mode to 0666. Aborted runs may leave the files with mode 0222 and the *uulog* program will not read or remove them. To remove them, either use *rm*, *uuclean*, or change the mode to 0666 and let *uulog* merge them with the *LOGFILE*.

STST — system status files

These files are created in the spool directory by the *uucico* program. They contain information of failures such as login, dialup or sequence check and will contain a *TALKING* status when to machines are conversing. The form of the file name is

        STST.sys

where *sys* is the remote system name.

For ordinary failures (dialup, login), the file will prevent repeated tries for about one hour. For sequence check failures, the file must be removed before any future attempts to converse with that remote system.

If the file is left due to an aborted run, it may contain a *TALKING* status. In this case, the file must be removed before a conversation is attempted.

LCK — lock files

Lock files are created for each device in use (e.g. automatic calling unit) and each system conversing. This prevents duplicate conversations and multiple attempts to use the same devices. The form of the lock file name is

        LCK..str

where *str* is either a device or system name. The files may be left in the spool directory if runs abort. They will be ignored (reused) after a time of about 24 hours. When runs abort and calls are desired before the time limit, the lock files should be removed.

Shell Files

The *uucp* program will spool work and attempt to start the *uucico* program, but the starting of *uucico* will sometimes fail. (No devices available, login failures etc.). Therefore, the *uucico* program should be periodically started. The command to start *uucico* can be put in a "shell" file with a command to merge *LOG* files and started by a crontab entry on an hourly basis. The file could contain the commands

        *program/*uulog
        *program/*uucico   −rl

Note that the "−rl" option is required to start the *uucico* program in *MASTER* mode.

Another shell file may be set up on a daily basis to remove *TM*, *ST* and *LCK* files and *C.* or *D.* files for work which can not be accomplished for reasons like bad phone number, login changes etc. A shell file containing commands like

```
program/uuclean    −pTM −pC. −pD.
program/uuclean    −pST −pLCK −n12
```

can be used. Note the "−n12" option causes the *ST* and *LCK* files older than 12 hours to be deleted. The absence of the "−n" option will use a three day time limit.

A daily or weekly shell should also be created to remove or save old *LOGFILE*s. A shell like

```
cp spool/LOGFILE    spool/o.LOGFILE
rm spool/LOGFILE
```

can be used.

Login Entry

One or more logins should be set up for *uucp.* Each of the "/etc/passwd" entries should have the "*program/*uucico" as the shell to be executed. The login directory is not used, but if the system has a special directory for use by the users for sending or receiving file, it should as the login entry. The various logins are used in conjunction with the *USERFILE* to restrict file access. Specifying the *shell* argument limits the login to the use of uucp (*uucico*) only.

File Modes

It is suggested that the owner and file modes of various programs and files be set as follows.

The programs *uucp, uux, uucico* and *uuxqt* should be owned by the *uucp* login with the "setuid" bit set and only execute permissions (e.g. mode 04111). This will prevent outsiders from modifying the programs to get at a standard *shell* for the *uucp* logins.

The *L.sys, SQFILE* and the *USERFILE* which are put in the *program* directory should be owned by the *uucp* login and set with mode 0400.

UNIX Implementation

K. Thompson

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes in high-level terms the implementation of the resident UNIX† kernel. This discussion is broken into three parts. The first part describes how the UNIX system views processes, users, and programs. The second part describes the I/O system. The last part describes the UNIX file system.

1. INTRODUCTION

The UNIX kernel consists of about 10,000 lines of C code and about 1,000 lines of assembly code. The assembly code can be further broken down into 200 lines included for the sake of efficiency (they could have been written in C) and 800 lines to perform hardware functions not possible in C.

This code represents 5 to 10 percent of what has been lumped into the broad expression "the UNIX operating system." The kernel is the only UNIX code that cannot be substituted by a user to his own liking. For this reason, the kernel should make as few real decisions as possible. This does not mean to allow the user a million options to do the same thing. Rather, it means to allow only one way to do one thing, but have that way be the least-common divisor of all the options that might have been provided.

What is or is not implemented in the kernel represents both a great responsibility and a great power. It is a soap-box platform on "the way things should be done." Even so, if "the way" is too radical, no one will follow it. Every important decision was weighed carefully. Throughout, simplicity has been substituted for efficiency. Complex algorithms are used only if their complexity can be localized.

2. PROCESS CONTROL

In the UNIX system, a user executes programs in an environment called a user process. When a system function is required, the user process calls the system as a subroutine. At some point in this call, there is a distinct switch of environments. After this, the process is said to be a system process. In the normal definition of processes, the user and system processes are different phases of the same process (they never execute simultaneously). For protection, each system process has its own stack.

The user process may execute from a read-only text segment, which is shared by all processes executing the same code. There is no *functional* benefit from shared-text segments. An *efficiency* benefit comes from the fact that there is no need to swap read-only segments out because the original copy on secondary memory is still current. This is a great benefit to interactive programs that tend to be swapped while waiting for terminal input. Furthermore, if two processes are executing simultaneously from the same copy of a read-only segment, only one copy needs to reside in primary memory. This is a secondary effect, because simultaneous

---

†UNIX is a Trademark of Bell Laboratories.

execution of a program is not common. It is ironic that this effect, which reduces the use of primary memory, only comes into play when there is an overabundance of primary memory, that is, when there is enough memory to keep waiting processes loaded.

All current read-only text segments in the system are maintained from the *text table*. A text table entry holds the location of the text segment on secondary memory. If the segment is loaded, that table also holds the primary memory location and the count of the number of processes sharing this entry. When this count is reduced to zero, the entry is freed along with any primary and secondary memory holding the segment. When a process first executes a shared-text segment, a text table entry is allocated and the segment is loaded onto secondary memory. If a second process executes a text segment that is already allocated, the entry reference count is simply incremented.

A user process has some strictly private read-write data contained in its data segment. As far as possible, the system does not use the user's data segment to hold system data. In particular, there are no I/O buffers in the user address space.

The user data segment has two growing boundaries. One, increased automatically by the system as a result of memory faults, is used for a stack. The second boundary is only grown (or shrunk) by explicit requests. The contents of newly allocated primary memory is initialized to zero.

Also associated and swapped with a process is a small fixed-size system data segment. This segment contains all the data about the process that the system needs only when the process is active. Examples of the kind of data contained in the system data segment are: saved central processor registers, open file descriptors, accounting information, scratch data area, and the stack for the system phase of the process. The system data segment is not addressable from the user process and is therefore protected.

Last, there is a process table with one entry per process. This entry contains all the data needed by the system when the process is *not* active. Examples are the process's name, the location of the other segments, and scheduling information. The process table entry is allocated when the process is created, and freed when the process terminates. This process entry is always directly addressable by the kernel.

Figure 1 shows the relationships between the various process control data. In a sense, the process table is the definition of all processes, because all the data associated with a process may be accessed starting from the process table entry.
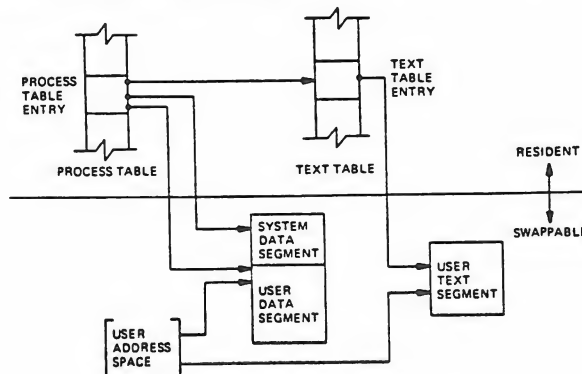
Fig. 1—Process control data structure.

2.1. Process creation and program execution

Processes are created by the system primitive **fork**. The newly created process (child) is a copy of the original process (parent). There is no detectable sharing of primary memory between the two processes. (Of course, if the parent process was executing from a read-only text segment, the child will share the text segment.) Copies of all writable data segments are made for the child process. Files that were open before the **fork** are truly shared after the **fork**. The processes are informed as to their part in the relationship to allow them to select their own (usually non-identical) destiny. The parent may **wait** for the termination of any of its children.

A process may **exec** a file. This consists of exchanging the current text and data segments of the process for new text and data segments specified in the file. The old segments are lost. Doing an **exec** does *not* change processes; the process that did the **exec** persists, but after the **exec** it is executing a different program. Files that were open before the **exec** remain open after the **exec**.

If a program, say the first pass of a compiler, wishes to overlay itself with another program, say the second pass, then it simply **execs** the second program. This is analogous to a "goto." If a program wishes to regain control after **exec**ing a second program, it should **fork** a child process, have the child **exec** the second program, and have the parent **wait** for the child. This is analogous to a "call." Breaking up the call into a binding followed by a transfer is similar to the subroutine linkage in SL-5.[1]

2.2. Swapping

The major data associated with a process (the user data segment, the system data segment, and the text segment) are swapped to and from secondary memory, as needed. The user data segment and the system data segment are kept in contiguous primary memory to reduce swapping latency. (When low-latency devices, such as bubbles, CCDs, or scatter/gather devices, are used, this decision will have to be reconsidered.) Allocation of both primary and secondary memory is performed by the same simple first-fit algorithm. When a process grows, a new piece of primary memory is allocated. The contents of the old memory is copied to the new memory. The old memory is freed and the tables are updated. If there is not enough primary memory, secondary memory is allocated instead. The process is swapped out onto the secondary memory, ready to be swapped in with its new size.

One separate process in the kernel, the swapping process, simply swaps the other processes in and out of primary memory. It examines the process table looking for a process that is swapped out and is ready to run. It allocates primary memory for that process and reads its segments into primary memory, where that process competes for the central processor with other loaded processes. If no primary memory is available, the swapping process makes memory available by examining the process table for processes that can be swapped out. It selects a process to swap out, writes it to secondary memory, frees the primary memory, and then goes back to look for a process to swap in.

Thus there are two specific algorithms to the swapping process. Which of the possibly many processes that are swapped out is to be swapped in? This is decided by secondary storage residence time. The one with the longest time out is swapped in first. There is a slight penalty for larger processes. Which of the possibly many processes that are loaded is to be swapped out? Processes that are waiting for slow events (i.e., not currently running or waiting for disk I/O) are picked first, by age in primary memory, again with size penalties. The other processes are examined by the same age algorithm, but are not taken out unless they are at least of some age. This adds hysteresis to the swapping and prevents total thrashing.

These swapping algorithms are the most suspect in the system. With limited primary memory, these algorithms cause total swapping. This is not bad in itself, because the swapping does not impact the execution of the resident processes. However, if the swapping device must also be used for file storage, the swapping traffic severely impacts the file system traffic. It is exactly these small systems that tend to double usage of limited disk resources.

2.3. Synchronization and scheduling

Process synchronization is accomplished by having processes wait for events. Events are represented by arbitrary integers. By convention, events are chosen to be addresses of tables associated with those events. For example, a process that is waiting for any of its children to terminate will wait for an event that is the address of its own process table entry. When a process terminates, it signals the event represented by its parent's process table entry. Signaling an event on which no process is waiting has no effect. Similarly, signaling an event on which many processes are waiting will wake all of them up. This differs considerably from Dijkstra's P and V synchronization operations,[2] in that no memory is associated with events. Thus there need be no allocation of events prior to their use. Events exist simply by being used.

On the negative side, because there is no memory associated with events, no notion of "how much" can be signaled via the event mechanism. For example, processes that want memory might wait on an event associated with memory allocation. When any amount of memory becomes available, the event would be signaled. All the competing processes would then wake up to fight over the new memory. (In reality, the swapping process is the only process that waits for primary memory to become available.)

If an event occurs between the time a process decides to wait for that event and the time that process enters the wait state, then the process will wait on an event that has already happened (and may never happen again). This race condition happens because there is no memory associated with the event to indicate that the event has occurred; the only action of an event is to change a set of processes from wait state to run state. This problem is relieved largely by the fact that process switching can only occur in the kernel by explicit calls to the event-wait mechanism. If the event in question is signaled by another process, then there is no problem. But if the event is signaled by a hardware interrupt, then special care must be taken. These synchronization races pose the biggest problem when UNIX is adapted to multiple-processor configurations.[3]

The event-wait code in the kernel is like a co-routine linkage. At any time, all but one of the processes has called event-wait. The remaining process is the one currently executing. When it calls event-wait, a process whose event has been signaled is selected and that process returns from its call to event-wait.

Which of the runable processes is to run next? Associated with each process is a priority. The priority of a system process is assigned by the code issuing the wait on an event. This is roughly equivalent to the response that one would expect on such an event. Disk events have high priority, teletype events are low, and time-of-day events are very low. (From observation, the difference in system process priorities has little or no performance impact.) All user-process priorities are lower than the lowest system priority. User-process priorities are assigned by an algorithm based on the recent ratio of the amount of compute time to real time consumed by the process. A process that has used a lot of compute time in the last real-time unit is assigned a low user priority. Because interactive processes are characterized by low ratios of compute to real time, interactive response is maintained without any special arrangements.

The scheduling algorithm simply picks the process with the highest priority, thus picking all system processes first and user processes second. The compute-to-real-time ratio is updated every second. Thus, all other things being equal, looping user processes will be scheduled round-robin with a 1-second quantum. A high-priority process waking up will preempt a running, low-priority process. The scheduling algorithm has a very desirable negative feedback character. If a process uses its high priority to hog the computer, its priority will drop. At the same time, if a low-priority process is ignored for a long time, its priority will rise.

3. I/O SYSTEM

The I/O system is broken into two completely separate systems: the block I/O system and the character I/O system. In retrospect, the names should have been "structured I/O" and "unstructured I/O," respectively; while the term "block I/O" has some meaning, "character

I/O" is a complete misnomer.

Devices are characterized by a major device number, a minor device number, and a class (block or character). For each class, there is an array of entry points into the device drivers. The major device number is used to index the array when calling the code for a particular device driver. The minor device number is passed to the device driver as an argument. The minor number has no significance other than that attributed to it by the driver. Usually, the driver uses the minor number to access one of several identical physical devices.

The use of the array of entry points (configuration table) as the only connection between the system code and the device drivers is very important. Early versions of the system had a much less formal connection with the drivers, so that it was extremely hard to handcraft differently configured systems. Now it is possible to create new device drivers in an average of a few hours. The configuration table in most cases is created automatically by a program that reads the system's parts list.

3.1. Block I/O system

The model block I/O device consists of randomly addressed, secondary memory blocks of 512 bytes each. The blocks are uniformly addressed 0, 1, ... up to the size of the device. The block device driver has the job of emulating this model on a physical device.

The block I/O devices are accessed through a layer of buffering software. The system maintains a list of buffers (typically between 10 and 70) each assigned a device name and a device address. This buffer pool constitutes a data cache for the block devices. On a read request, the cache is searched for the desired block. If the block is found, the data are made available to the requester without any physical I/O. If the block is not in the cache, the least recently used block in the cache is renamed, the correct device driver is called to fill up the renamed buffer, and then the data are made available. Write requests are handled in an analogous manner. The correct buffer is found and relabeled if necessary. The write is performed simply by marking the buffer as "dirty." The physical I/O is then deferred until the buffer is renamed.

The benefits in reduction of physical I/O of this scheme are substantial, especially considering the file system implementation. There are, however, some drawbacks. The asynchronous nature of the algorithm makes error reporting and meaningful user error handling almost impossible. The cavalier approach to I/O error handling in the UNIX system is partly due to the asynchronous nature of the block I/O system. A second problem is in the delayed writes. If the system stops unexpectedly, it is almost certain that there is a lot of logically complete, but physically incomplete, I/O in the buffers. There is a system primitive to flush all outstanding I/O activity from the buffers. Periodic use of this primitive helps, but does not solve, the problem. Finally, the associativity in the buffers can alter the physical I/O sequence from that of the logical I/O sequence. This means that there are times when data structures on disk are inconsistent, even though the software is careful to perform I/O in the correct order. On nonrandom devices, notably magnetic tape, the inversions of writes can be disastrous. The problem with magnetic tapes is "cured" by allowing only one outstanding write request per drive.

3.2. Character I/O system

The character I/O system consists of all devices that do not fall into the block I/O model. This includes the "classical" character devices such as communications lines, paper tape, and line printers. It also includes magnetic tape and disks when they are not used in a stereotyped way, for example, 80-byte physical records on tape and track-at-a-time disk copies. In short, the character I/O interface means "everything other than block." I/O requests from the user are sent to the device driver essentially unaltered. The implementation of these requests is, of course, up to the device driver. There are guidelines and conventions to help the implementation of certain types of device drivers.

3.2.1. Disk drivers

Disk drivers are implemented with a queue of transaction records. Each record holds a read/write flag, a primary memory address, a secondary memory address, and a transfer byte count. Swapping is accomplished by passing such a record to the swapping device driver. The block I/O interface is implemented by passing such records with requests to fill and empty system buffers. The character I/O interface to the disk drivers create a transaction record that points directly into the user area. The routine that creates this record also insures that the user is not swapped during this I/O transaction. Thus by implementing the general disk driver, it is possible to use the disk as a block device, a character device, and a swap device. The only really disk-specific code in normal disk drivers is the pre-sort of transactions to minimize latency for a particular device, and the actual issuing of the I/O request.

3.2.2. Character lists

Real character-oriented devices may be implemented using the common code to handle character lists. A character list is a queue of characters. One routine puts a character on a queue. Another gets a character from a queue. It is also possible to ask how many characters are currently on a queue. Storage for all queues in the system comes from a single common pool. Putting a character on a queue will allocate space from the common pool and link the character onto the data structure defining the queue. Getting a character from a queue returns the corresponding space to the pool.

A typical character-output device (paper tape punch, for example) is implemented by passing characters from the user onto a character queue until some maximum number of characters is on the queue. The I/O is prodded to start as soon as there is anything on the queue and, once started, it is sustained by hardware completion interrupts. Each time there is a completion interrupt, the driver gets the next character from the queue and sends it to the hardware. The number of characters on the queue is checked and, as the count falls through some intermediate level, an event (the queue address) is signaled. The process that is passing characters from the user to the queue can be waiting on the event, and refill the queue to its maximum when the event occurs.

A typical character input device (for example, a paper tape reader) is handled in a very similar manner.

Another class of character devices is the terminals. A terminal is represented by three character queues. There are two input queues (raw and canonical) and an output queue. Characters going to the output of a terminal are handled by common code exactly as described above. The main difference is that there is also code to interpret the output stream as ASCII characters and to perform some translations, e.g., escapes for deficient terminals. Another common aspect of terminals is code to insert real-time delay after certain control characters.

Input on terminals is a little different. Characters are collected from the terminal and placed on a raw input queue. Some device-dependent code conversion and escape interpretation is handled here. When a line is complete in the raw queue, an event is signaled. The code catching this signal then copies a line from the raw queue to a canonical queue performing the character erase and line kill editing. User read requests on terminals can be directed at either the raw or canonical queues.

3.2.3. Other character devices

Finally, there are devices that fit no general category. These devices are set up as character I/O drivers. An example is a driver that reads and writes unmapped primary memory as an I/O device. Some devices are too fast to be treated a character at time, but do not fit the disk I/O mold. Examples are fast communications lines and fast line printers. These devices either have their own buffers or "borrow" block I/O buffers for a while and then give them back.

4. THE FILE SYSTEM

In the UNIX system, a file is a (one-dimensional) array of bytes. No other structure of files is implied by the system. Files are attached anywhere (and possibly multiply) onto a hierarchy of directories. Directories are simply files that users cannot write. For a further discussion of the external view of files and directories, see Ref. 4.

The UNIX file system is a disk data structure accessed completely through the block I/O system. As stated before, the canonical view of a "disk" is a randomly addressable array of 512-byte blocks. A file system breaks the disk into four self-identifying regions. The first block (address 0) is unused by the file system. It is left aside for booting procedures. The second block (address 1) contains the so-called "super-block." This block, among other things, contains the size of the disk and the boundaries of the other regions. Next comes the i-list, a list of file definitions. Each file definition is a 64-byte structure, called an i-node. The offset of a particular i-node within the i-list is called its i-number. The combination of device name (major and minor numbers) and i-number serves to uniquely name a particular file. After the i-list, and to the end of the disk, come free storage blocks that are available for the contents of files.

The free space on a disk is maintained by a linked list of available disk blocks. Every block in this chain contains a disk address of the next block in the chain. The remaining space contains the address of up to 50 disk blocks that are also free. Thus with one I/O operation, the system obtains 50 free blocks and a pointer where to find more. The disk allocation algorithms are very straightforward. Since all allocation is in fixed-size blocks and there is strict accounting of space, there is no need to compact or garbage collect. However, as disk space becomes dispersed, latency gradually increases. Some installations choose to occasionally compact disk space to reduce latency.

An i-node contains 13 disk addresses. The first 10 of these addresses point directly at the first 10 blocks of a file. If a file is larger than 10 blocks (5,120 bytes), then the eleventh address points at a block that contains the addresses of the next 128 blocks of the file. If the file is still larger than this (70,656 bytes), then the twelfth block points at up to 128 blocks, each pointing to 128 blocks of the file. Files yet larger (8,459,264 bytes) use the thirteenth address for a "triple indirect" address. The algorithm ends here with the maximum file size of 1,082,201,087 bytes.

A logical directory hierarchy is added to this flat physical structure simply by adding a new type of file, the directory. A directory is accessed exactly as an ordinary file. It contains 16-byte entries consisting of a 14-byte name and an i-number. The root of the hierarchy is at a known i-number (viz., 2). The file system structure allows an arbitrary, directed graph of directories with regular files linked in at arbitrary places in this graph. In fact, very early UNIX systems used such a structure. Administration of such a structure became so chaotic that later systems were restricted to a directory tree. Even now, with regular files linked multiply into arbitrary places in the tree, accounting for space has become a problem. It may become necessary to restrict the entire structure to a tree, and allow a new form of linking that is subservient to the tree structure.

The file system allows easy creation, easy removal, easy random accessing, and very easy space allocation. With most physical addresses confined to a small contiguous section of disk, it is also easy to dump, restore, and check the consistency of the file system. Large files suffer from indirect addressing, but the cache prevents most of the implied physical I/O without adding much execution. The space overhead properties of this scheme are quite good. For example, on one particular file system, there are 25,000 files containing 130M bytes of data-file content. The overhead (i-node, indirect blocks, and last block breakage) is about 11.5M bytes. The directory structure to support these files has about 1,500 directories containing 0.6M bytes of directory content and about 0.5M bytes of overhead in accessing the directories. Added up any way, this comes out to less than a 10 percent overhead for actual stored data. Most systems have this much overhead in padded trailing blanks alone.

4.1. File system implementation

Because the i-node defines a file, the implementation of the file system centers around access to the i-node. The system maintains a table of all active i-nodes. As a new file is accessed, the system locates the corresponding i-node, allocates an i-node table entry, and reads the i-node into primary memory. As in the buffer cache, the table entry is considered to be the current version of the i-node. Modifications to the i-node are made to the table entry. When the last access to the i-node goes away, the table entry is copied back to the secondary store i-list and the table entry is freed.

All I/O operations on files are carried out with the aid of the corresponding i-node table entry. The accessing of a file is a straightforward implementation of the algorithms mentioned previously. The user is not aware of i-nodes and i-numbers. References to the file system are made in terms of path names of the directory tree. Converting a path name into an i-node table entry is also straightforward. Starting at some known i-node (the root or the current directory of some process), the next component of the path name is searched by reading the directory. This gives an i-number and an implied device (that of the directory). Thus the next i-node table entry can be accessed. If that was the last component of the path name, then this i-node is the result. If not, this i-node is the directory needed to look up the next component of the path name, and the algorithm is repeated.

The user process accesses the file system with certain primitives. The most common of these are open, create, read, write, seek, and close. The data structures maintained are shown in Fig. 2.
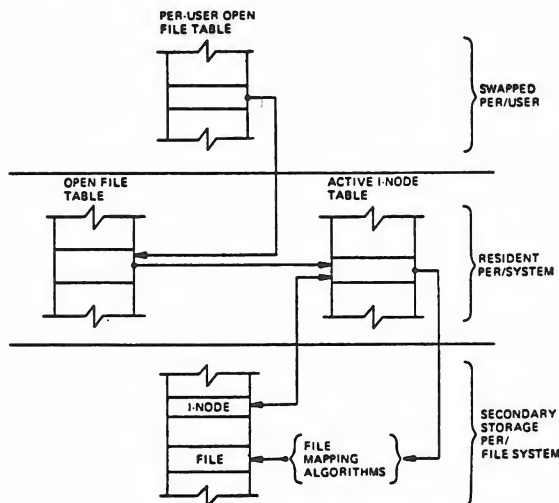
Fig. 2—File system data structure.

In the system data segment associated with a user, there is room for some (usually between 10 and 50) open files. This open file table consists of pointers that can be used to access corresponding i-node table entries. Associated with each of these open files is a current I/O pointer. This is a byte offset of the next read/write operation on the file. The system treats each read/write request as random with an implied seek to the I/O pointer. The user usually thinks of the file as sequential with the I/O pointer automatically counting the number of bytes that have been read/written from the file. The user may, of course, perform random I/O by setting the I/O pointer before reads/writes.

With file sharing, it is necessary to allow related processes to share a common I/O pointer

and yet have separate I/O pointers for independent processes that access the same file. With these two conditions, the I/O pointer cannot reside in the i-node table nor can it reside in the list of open files for the process. A new table (the open file table) was invented for the sole purpose of holding the I/O pointer. Processes that share the same open file (the result of **forks**) share a common open file table entry. A separate open of the same file will only share the i-node table entry, but will have distinct open file table entries.

The main file system primitives are implemented as follows. **open** converts a file system path name into an i-node table entry. A pointer to the i-node table entry is placed in a newly created open file table entry. A pointer to the file table entry is placed in the system data segment for the process. **create** first creates a new i-node entry, writes the i-number into a directory, and then builds the same structure as for an **open**. **read** and **write** just access the i-node entry as described above. **seek** simply manipulates the I/O pointer. No physical seeking is done. **close** just frees the structures built by **open** and **create**. Reference counts are kept on the open file table entries and the i-node table entries to free these structures after the last reference goes away. **unlink** simply decrements the count of the number of directories pointing at the given i-node. When the last reference to an i-node table entry goes away, if the i-node has no directories pointing to it, then the file is removed and the i-node is freed. This delayed removal of files prevents problems arising from removing active files. A file may be removed while still open. The resulting unnamed file vanishes when the file is closed. This is a method of obtaining temporary files.

There is a type of unnamed FIFO file called a **pipe.** Implementation of **pipes** consists of implied **seeks** before each **read** or **write** in order to implement first-in-first-out. There are also checks and synchronization to prevent the writer from grossly outproducing the reader and to prevent the reader from overtaking the writer.

4.2. Mounted file systems

The file system of a UNIX system starts with some designated block device formatted as described above to contain a hierarchy. The root of this structure is the root of the UNIX file system. A second formatted block device may be mounted at any leaf of the current hierarchy This logically extends the current hierarchy. The implementation of mounting is trivial. A mount table is maintained containing pairs of designated leaf i-nodes and block devices. When converting a path name into an i-node, a check is made to see if the new i-node is a designated leaf. If it is, the i-node of the root of the block device replaces it.

Allocation of space for a file is taken from the free pool on the device on which the file lives. Thus a file system consisting of many mounted devices does not have a common pool of free secondary storage space. This separation of space on different devices is necessary to allow easy unmounting of a device.

4.3. Other system functions

There are some other things that the system does for the user—a little accounting, a little tracing/debugging, and a little access protection. Most of these things are not very well developed because our use of the system in computing science research does not need them. There are some features that are missed in some applications, for example, better inter-process communication.

The UNIX kernel is an I/O multiplexer more than a complete operating system. This is as it should be. Because of this outlook, many features are found in most other operating systems that are missing from the UNIX kernel. For example, the UNIX kernel does not support file access methods, file disposition, file formats, file maximum size, spooling, command language, logical records, physical records, assignment of logical file names, logical file names, more than one character set, an operator's console, an operator, log-in, or log-out. Many of these things are symptoms rather than features. Many of these things are implemented in user software using the kernel as a tool. A good example of this is the command language.[5] Each user may have his own command language. Maintenance of such code is as easy as maintaining user

code. The idea of implementing "system" code with general user primitives comes directly from MULTICS.[6]

References

1. R. E. Griswold and D. R. Hanson, "An Overview of SL5," *SIGPLAN Notices* 12(4) pp. 40-50 (April 1977).

2. E. W. Dijkstra, "Cooperating Sequential Processes," pp. 43-112 in *Programming Languages*, ed. F. Genuys,Academic Press, New York (1968).

3. J. A. Hawley and W. B. Meyer, "MUNIX, A Multiprocessing Version of UNIX," M.S. Thesis, Naval Postgraduate School, Monterey, Cal. (1975).

4. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.* 57(6) pp. 1905-1929 (1978).

5. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," *Bell Sys. Tech. J.* 57(6) pp. 1971-1990 (1978).

6. E. I. Organick, *The MULTICS System*, M.I.T. Press, Cambridge, Mass. (1972).

The UNIX I/O System

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

This paper gives an overview of the workings of the UNIX† I/O system. It was written with an eye toward providing guidance to writers of device driver routines, and is oriented more toward describing the environment and nature of device drivers than the implementation of that part of the file system which deals with ordinary files.

It is assumed that the reader has a good knowledge of the overall structure of the file system as discussed in the paper "The UNIX Time-sharing System." A more detailed discussion appears in "UNIX Implementation;" the current document restates parts of that one, but is still more detailed. It is most useful in conjunction with a copy of the system code, since it is basically an exegesis of that code.

Device Classes

There are two classes of device: *block* and *character*. The block interface is suitable for devices like disks, tapes, and DECtape which work, or can work, with addressible 512-byte blocks. Ordinary magnetic tape just barely fits in this category, since by use of forward and backward spacing any block can be read, even though blocks can be written only at the end of the tape. Block devices can at least potentially contain a mounted file system. The interface to block devices is very highly structured; the drivers for these devices share a great many routines as well as a pool of buffers.

Character-type devices have a much more straightforward interface, although more work must be done by the driver itself.

Devices of both types are named by a *major* and a *minor* device number. These numbers are generally stored as an integer with the minor device number in the low-order 8 bits and the major device number in the next-higher 8 bits; macros *major* and *minor* are available to access these numbers. The major device number selects which driver will deal with the device; the minor device number is not used by the rest of the system but is passed to the driver at appropriate times. Typically the minor number selects a subdevice attached to a given controller, or one of several similar hardware interfaces.

The major device numbers for block and character devices are used as indices in separate tables; they both start at 0 and therefore overlap.

Overview of I/O

The purpose of the *open* and *creat* system calls is to set up entries in three separate system tables. The first of these is the *u_ofile* table, which is stored in the system's per-process data area *u*. This table is indexed by the file descriptor returned by the *open* or *creat*, and is accessed during a *read, write,* or other operation on the open file. An entry contains only a pointer to the corresponding entry of the *file* table, which is a per-system data base. There is one entry in the *file* table for each instance of *open* or *creat*. This table is per-system because the same instance of an open file must be shared among the several processes which can result from *forks* after

---

the file is opened. A *file* table entry contains flags which indicate whether the file was open for reading or writing or is a pipe, and a count which is used to decide when all processes using the entry have terminated or closed the file (so the entry can be abandoned). There is also a 32-bit file offset which is used to indicate where in the file the next read or write will take place. Finally, there is a pointer to the entry for the file in the *inode* table, which contains a copy of the file's i-node.

Certain open. files can be designated "multiplexed" files, and several other flags apply to such channels. In such a case, instead of an offset, there is a pointer to an associated multiplex channel table. Multiplex channels will not be discussed here.

An entry in the *file* table corresponds precisely to an instance of *open* or *creat;* if the same file is opened several times, it will have several entries in this table. However, there is at most one entry in the *inode* table for a given file. Also, a file may enter the *inode* table not only because it is open, but also because it is the current directory of some process or because it is a special file containing a currently-mounted file system.

An entry in the *inode* table differs somewhat from the corresponding i-node as stored on the disk; the modified and accessed times are not stored, and the entry is augmented by a flag word containing information about the entry, a count used to determine when it may be allowed to disappear, and the device and i-number whence the entry came. Also, the several block numbers that give addressing information for the file are expanded from the 3-byte, compressed format used on the disk to full *long* quantities.

During the processing of an *open* or *creat* call for a special file, the system always calls the device's *open* routine to allow for any special processing required (rewinding a tape, turning on the data-terminal-ready lead of a modem, etc.). However, the *close* routine is called only when the last process closes a file, that is, when the i-node table entry is being deallocated. Thus it is not feasible for a device to maintain, or depend on, a count of its users, although it is quite possible to implement an exclusive-use device which cannot be reopened until it has been closed.

When a *read* or *write* takes place, the user's arguments and the *file* table entry are used to set up the variables *u.u_base, u.u_count,* and *u.u_offset* which respectively contain the (user) address of the I/O target area, the byte-count for the transfer, and the current location in the file. If the file referred to is a character-type special file, the appropriate read or write routine is called; it is responsible for transferring data and updating the count and current location appropriately as discussed below. Otherwise, the current location is used to calculate a logical block number in the file. If the file is an ordinary file the logical block number must be mapped (possibly using indirect blocks) to a physical block number; a block-type special file need not be mapped. This mapping is performed by the *bmap* routine. In any event, the resulting physical block number is used, as discussed below, to read or write the appropriate device.

Character Device Drivers

The *cdevsw* table specifies the interface routines present for character devices. Each device provides five routines: open, close, read, write, and special-function (to implement the *ioctl* system call). Any of these may be missing. If a call on the routine should be ignored, (e.g. *open* on non-exclusive devices that require no setup) the *cdevsw* entry can be given as *nulldev;* if it should be considered an error, (e.g. *write* on read-only devices) *nodev* is used. For terminals, the *cdevsw* structure also contains a pointer to the *tty* structure associated with the terminal.

The *open* routine is called each time the file is opened with the full device number as argument. The second argument is a flag which is non-zero only if the device is to be written upon.

The *close* routine is called only when the file is closed for the last time, that is when the very last process in which the file is open closes it. This means it is not possible for the driver to maintain its own count of its users. The first argument is the device number; the second is a

flag which is non-zero if the file was open for writing in the process which performs the final *close*.

When *write* is called, it is supplied the device as argument. The per-user variable *u.u_count* has been set to the number of characters indicated by the user; for character devices, this number may be 0 initially. *u.u_base* is the address supplied by the user from which to start taking characters. The system may call the routine internally, so the flag *u.u_segflg* is supplied that indicates, if *on*, that *u.u_base* refers to the system address space instead of the user's.

The *write* routine should copy up to *u.u_count* characters from the user's buffer to the device, decrementing *u.u_count* for each character passed. For most drivers, which work one character at a time, the routine *cpass( )* is used to pick up characters from the user's buffer. Successive calls on it return the characters to be written until *u.u_count* goes to 0 or an error occurs, when it returns −1. *Cpass* takes care of interrogating *u.u_segflg* and updating *u.u_count*.

Write routines which want to transfer a probably large number of characters into an internal buffer may also use the routine *iomove(buffer, offset, count, flag)* which is faster when many characters must be moved. *Iomove* transfers up to *count* characters into the *buffer* starting *offset* bytes from the start of the buffer; *flag* should be *B_WRITE* (which is 0) in the write case. Caution: the caller is responsible for making sure the count is not too large and is non-zero. As an efficiency note, *iomove* is much slower if any of *buffer+offset*, *count* or *u.u_base* is odd.

The device's *read* routine is called under conditions similar to *write*, except that *u.u_count* is guaranteed to be non-zero. To return characters to the user, the routine *passc(c)* is available; it takes care of housekeeping like *cpass* and returns −1 as the last character specified by *u.u_count* is returned to the user; before that time, 0 is returned. *Iomove* is also usable as with *write;* the flag should be *B_READ* but the same cautions apply.

The "special-functions" routine is invoked by the *stty* and *gtty* system calls as follows: *(*p) (dev, v)* where *p* is a pointer to the device's routine, *dev* is the device number, and *v* is a vector. In the *gtty* case, the device is supposed to place up to 3 words of status information into the vector; this will be returned to the caller. In the *stty* case, *v* is 0; the device should take up to 3 words of control information from the array *u.u_arg[0...2]*.

Finally, each device should have appropriate interrupt-time routines. When an interrupt occurs, it is turned into a C-compatible call on the devices's interrupt routine. The interrupt-catching mechanism makes the low-order four bits of the "new PS" word in the trap vector for the interrupt available to the interrupt handler. This is conventionally used by drivers which deal with multiple similar devices to encode the minor device number. After the interrupt has been processed, a return from the interrupt handler will return from the interrupt itself.

A number of subroutines are available which are useful to character device drivers. Most of these handlers, for example, need a place to buffer characters in the internal interface between their "top half" (read/write) and "bottom half" (interrupt) routines. For relatively low data-rate devices, the best mechanism is the character queue maintained by the routines *getc* and *putc*. A queue header has the structure

```
struct {
        int     c_cc;   /* character count */
        char    *c_cf;  /* first character */
        char    *c_cl;  /* last character */
} queue;
```

A character is placed on the end of a queue by *putc(c, &queue)* where *c* is the character and *queue* is the queue header. The routine returns −1 if there is no space to put the character, 0 otherwise. The first character on the queue may be retrieved by *getc(&queue)* which returns either the (non-negative) character or −1 if the queue is empty.

Notice that the space for characters in queues is shared among all devices in the system and in the standard system there are only some 600 character slots available. Thus device handlers, especially write routines, must take care to avoid gobbling up excessive numbers of

characters.

The other major help available to device handlers is the sleep-wakeup mechanism. The call *sleep(event, priority)* causes the process to wait (allowing other processes to run) until the *event* occurs; at that time, the process is marked ready-to-run and the call will return when there is no process with higher *priority*.

The call *wakeup(event)* indicates that the *event* has happened, that is, causes processes sleeping on the event to be awakened. The *event* is an arbitrary quantity agreed upon by the sleeper and the waker-up. By convention, it is the address of some data area used by the driver, which guarantees that events are unique.

Processes sleeping on an event should not assume that the event has really happened; they should check that the conditions which caused them to sleep no longer hold.

Priorities can range from 0 to 127; a higher numerical value indicates a less-favored scheduling situation. A distinction is made between processes sleeping at priority less than the parameter *PZERO* and those at numerically larger priorities. The former cannot be interrupted by signals, although it is conceivable that it may be swapped out. Thus it is a bad idea to sleep with priority less than PZERO on an event which might never occur. On the other hand, calls to *sleep* with larger priority may never return if the process is terminated by some signal in the meantime. Incidentally, it is a gross error to call *sleep* in a routine called at interrupt time, since the process which is running is almost certainly not the process which should go to sleep. Likewise, none of the variables in the user area "*u.*" should be touched, let alone changed, by an interrupt routine.

If a device driver wishes to wait for some event for which it is inconvenient or impossible to supply a *wakeup*, (for example, a device going on-line, which does not generally cause an interrupt), the call *sleep(&lbolt, priority)* may be given. *Lbolt* is an external cell whose address is awakened once every 4 seconds by the clock interrupt routine.

The routines *spl4( ), spl5( ), spl6( ), spl7( )* are available to set the processor priority level as indicated to avoid inconvenient interrupts from the device.

If a device needs to know about real-time intervals, then *timeout(func, arg, interval)* will be useful. This routine arranges that after *interval* sixtieths of a second, the *func* will be called with *arg* as argument, in the style *(*func)(arg)*. Timeouts are used, for example, to provide real-time delays after function characters like new-line and tab in typewriter output, and to terminate an attempt to read the 201 Dataphone *dp* if there is no response within a specified number of seconds. Notice that the number of sixtieths of a second is limited to 32767, since it must appear to be positive, and that only a bounded number of timeouts can be going on at once. Also, the specified *func* is called at clock-interrupt time, so it should conform to the requirements of interrupt routines in general.

The Block-device Interface

Handling of block devices is mediated by a collection of routines that manage a set of buffers containing the images of blocks of data on the various devices. The most important purpose of these routines is to assure that several processes that access the same block of the same device in multiprogrammed fashion maintain a consistent view of the data in the block. A secondary but still important purpose is to increase the efficiency of the system by keeping in-core copies of blocks that are being accessed frequently. The main data base for this mechanism is the table of buffers *buf*. Each buffer header contains a pair of pointers *(b_forw, b_back)* which maintain a doubly-linked list of the buffers associated with a particular block device, and a pair of pointers *(av_forw, av_back)* which generally maintain a doubly-linked list of blocks which are "free," that is, eligible to be reallocated for another transaction. Buffers that have I/O in progress or are busy for other purposes do not appear in this list. The buffer header also contains the device and block number to which the buffer refers, and a pointer to the actual storage associated with the buffer. There is a word count which is the negative of the number of words to be transferred to or from the buffer; there is also an error byte and a

residual word count used to communicate information from an I/O routine to its caller. Finally, there is a flag word with bits indicating the status of the buffer. These flags will be discussed below.

Seven routines constitute the most important part of the interface with the rest of the system. Given a device and block number, both *bread* and *getblk* return a pointer to a buffer header for the block; the difference is that *bread* is guaranteed to return a buffer actually containing the current data for the block, while *getblk* returns a buffer which contains the data in the block only if it is already in core (whether it is or not is indicated by the *B_DONE* bit; see below). In either case the buffer, and the corresponding device block, is made "busy," so that other processes referring to it are obliged to wait until it becomes free. *Getblk* is used, for example, when a block is about to be totally rewritten, so that its previous contents are not useful; still, no other process can be allowed to refer to the block until the new data is placed into it.

The *breada* routine is used to implement read-ahead. it is logically similar to *bread,* but takes as an additional argument the number of a block (on the same device) to be read asynchronously after the specifically requested block is available.

Given a pointer to a buffer, the *brelse* routine makes the buffer again available to other processes. It is called, for example, after data has been extracted following a *bread.* There are three subtly-different write routines, all of which take a buffer pointer as argument, and all of which logically release the buffer for use by others and place it on the free list. *Bwrite* puts the buffer on the appropriate device queue, waits for the write to be done, and sets the user's error flag if required. *Bawrite* places the buffer on the device's queue, but does not wait for completion, so that errors cannot be reflected directly to the user. *Bdwrite* does not start any I/O operation at all, but merely marks the buffer so that if it happens to be grabbed from the free list to contain data from some other block, the data in it will first be written out.

Bwrite is used when one wants to be sure that I/O takes place correctly, and that errors are reflected to the proper user; it is used, for example, when updating i-nodes. *Bawrite* is useful when more overlap is desired (because no wait is required for I/O to finish) but when it is reasonably certain that the write is really required. *Bdwrite* is used when there is doubt that the write is needed at the moment. For example, *bdwrite* is called when the last byte of a *write* system call falls short of the end of a block, on the assumption that another *write* will be given soon which will re-use the same block. On the other hand, as the end of a block is passed, *bawrite* is called, since probably the block will not be accessed again soon and one might as well start the writing process as soon as possible.

In any event, notice that the routines *getblk* and *bread* dedicate the given block exclusively to the use of the caller, and make others wait, while one of *brelse, bwrite, bawrite,* or *bdwrite* must eventually be called to free the block for use by others.

As mentioned, each buffer header contains a flag word which indicates the status of the buffer. Since they provide one important channel for information between the drivers and the block I/O system, it is important to understand these flags. The following names are manifest constants which select the associated flag bits.

B_READ   This bit is set when the buffer is handed to the device strategy routine (see below) to indicate a read operation. The symbol *B_WRITE* is defined as 0 and does not define a flag; it is provided as a mnemonic convenience to callers of routines like *swap* which have a separate argument which indicates read or write.

B_DONE   This bit is set to 0 when a block is handed to the the device strategy routine and is turned on when the operation completes, whether normally as the result of an error. It is also used as part of the return argument of *getblk* to indicate if 1 that the returned buffer actually contains the data in the requested block.

B_ERROR This bit may be set to 1 when *B_DONE* is set to indicate that an I/O or other error occurred. If it is set the *b_error* byte of the buffer header may contain an error code if it is non-zero. If *b_error* is 0 the nature of the error is not specified. Actually no driver at present sets *b_error;* the latter is provided for a future improvement whereby a more detailed error-reporting scheme may be implemented.

B_BUSY This bit indicates that the buffer header is not on the free list, i.e. is dedicated to someone's exclusive use. The buffer still remains attached to the list of blocks associated with its device, however. When *getblk* (or *bread,* which calls it) searches the buffer list for a given device and finds the requested block with this bit on, it sleeps until the bit clears.

B_PHYS This bit is set for raw I/O transactions that need to allocate the Unibus map on an 11/70.

B_MAP This bit is set on buffers that have the Unibus map allocated, so that the *iodone* routine knows to deallocate the map.

B_WANTED This flag is used in conjunction with the *B_BUSY* bit. Before sleeping as described just above, *getblk* sets this flag. Conversely, when the block is freed and the busy bit goes down (in *brelse)* a *wakeup* is given for the block header whenever *B_WANTED* is on. This strategem avoids the overhead of having to call *wakeup* every time a buffer is freed on the chance that someone might want it.

B_AGE This bit may be set on buffers just before releasing them; if it is on, the buffer is placed at the head of the free list, rather than at the tail. It is a performance heuristic used when the caller judges that the same block will not soon be used again.

B_ASYNC This bit is set by *bawrite* to indicate to the appropriate device driver that the buffer should be released when the write has been finished, usually at interrupt time. The difference between *bwrite* and *bawrite* is that the former starts I/O, waits until it is done, and frees the buffer. The latter merely sets this bit and starts I/O. The bit indicates that *relse* should be called for the buffer on completion.

B_DELWRI This bit is set by *bdwrite* before releasing the buffer. When *getblk,* while searching for a free block, discovers the bit is 1 in a buffer it would otherwise grab, it causes the block to be written out before reusing it.

Block Device Drivers

The *bdevsw* table contains the names of the interface routines and that of a table for each block device.

Just as for character devices, block device drivers may supply an *open* and a *close* routine called respectively on each open and on the final close of the device. Instead of separate read and write routines, each block device driver has a *strategy* routine which is called with a pointer to a buffer header as argument. As discussed, the buffer header contains a read/write flag, the core address, the block number, a (negative) word count, and the major and minor device number. The role of the strategy routine is to carry out the operation as requested by the information in the buffer header. When the transaction is complete the *B_DONE* (and possibly the *B_ERROR)* bits should be set. Then if the *B_ASYNC* bit is set, *brelse* should be called; otherwise, *wakeup*. In cases where the device is capable, under error-free operation, of transferring fewer words than requested, the device's word-count register should be placed in the residual count slot of the buffer header; otherwise, the residual count should be set to 0. This particular mechanism is really for the benefit of the magtape driver; when reading this device records shorter than requested are quite normal, and the user should be told the actual length of the record.

Although the most usual argument to the strategy routines is a genuine buffer header allocated as discussed above, all that is actually required is that the argument be a pointer to a place containing the appropriate information. For example the *swap* routine, which manages movement of core images to and from the swapping device, uses the strategy routine for this

device. Care has to be taken that no extraneous bits get turned on in the flag word.

The device's table specified by *bdevsw* has a byte to contain an active flag and an error count, a pair of links which constitute the head of the chain of buffers for the device *(b_forw, b_back)*, and a first and last pointer for a device queue. Of these things, all are used solely by the device driver itself except for the buffer-chain pointers. Typically the flag encodes the state of the device, and is used at a minimum to indicate that the device is currently engaged in transferring information and no new command should be issued. The error count is useful for counting retries when errors occur. The device queue is used to remember stacked requests; in the simplest case it may be maintained as a first-in first-out list. Since buffers which have been handed over to the strategy routines are never on the list of free buffers, the pointers in the buffer which maintain the free list *(av_forw, av_back)* are also used to contain the pointers which maintain the device queues.

A couple of routines are provided which are useful to block device drivers. *iodone(bp)* arranges that the buffer to which *bp* points be released or awakened, as appropriate, when the strategy module has finished with the buffer, either normally or after an error. (In the latter case the *B_ERROR* bit has presumably been set.)

The routine *geterror(bp)* can be used to examine the error bit in a buffer header and arrange that any error indication found therein is reflected to the user. It may be called only in the non-interrupt part of a driver when I/O has completed *(B_DONE* has been set).

Raw Block-device I/O

A scheme has been set up whereby block device drivers may provide the ability to transfer information directly between the user's core image and the device without the use of buffers and in blocks as large as the caller requests. The method involves setting up a character-type special file corresponding to the raw device and providing *read* and *write* routines which set up what is usually a private, non-shared buffer header with the appropriate information and call the device's strategy routine. If desired, separate *open* and *close* routines may be provided but this is usually unnecessary. A special-function routine might come in handy, especially for magtape.

A great deal of work has to be done to generate the "appropriate information" to put in the argument buffer for the strategy module; the worst part is to map relocated user addresses to physical addresses. Most of this work is done by *physio(strat, bp, dev, rw)* whose arguments are the name of the strategy routine *strat*, the buffer pointer *bp*, the device number *dev*, and a read-write flag *rw* whose value is either *B_READ* or *B_WRITE*. *Physio* makes sure that the user's base address and count are even (because most devices work in words) and that the core area affected is contiguous in physical space; it delays until the buffer is not busy, and makes it busy while the operation is in progress; and it sets up user error return information.

A Fast File System for UNIX*

Revised July 27, 1983

Marshall Kirk McKusick, William N. Joy†,
Samuel J. Leffler‡, Robert S. Fabry

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

A reimplementation of the UNIX file system is described. The reimplementation provides substantially higher throughput rates by using more flexible allocation policies, that allow better locality of reference and that can be adapted to a wide range of peripheral and processor characteristics. The new file system clusters data that is sequentially accessed and provides two block sizes to allow fast access for large files while not wasting large amounts of space for small files. File access rates of up to ten times faster than the traditional UNIX file system are experienced. Long needed enhancements to the user interface are discussed. These include a mechanism to lock files, extensions of the name space across file systems, the ability to use arbitrary length file names, and provisions for efficient administrative control of resource usage.

.

TABLE OF CONTENTS

1. Introduction

This paper describes the changes from the original 512 byte UNIX file system to the new one released with the 4.2 Berkeley Software Distribution. It presents the motivations for the changes, the methods used to affect these changes, the rationale behind the design decisions, and a description of the new implementation. This discussion is followed by a summary of the results that have been obtained, directions for future work, and the additions and changes that have been made to the user visible facilities. The paper concludes with a history of the software engineering of the project.

The original UNIX system that runs on the PDP-11† has simple and elegant file system facilities. File system input/output is buffered by the kernel; there are no alignment constraints on data transfers and all operations are made to appear synchronous. All transfers to the disk are in 512 byte blocks, which can be placed arbitrarily within the data area of the file system. No constraints other than available disk space are placed on file growth [Ritchie74], [Thompson79].

When used on the VAX-11 together with other UNIX enhancements, the original 512 byte UNIX file system is incapable of providing the data throughput rates that many applications require. For example, applications that need to do a small amount of processing on a large quantities of data such as VLSI design and image processing, need to have a high throughput from the file system. High throughput rates are also needed by programs with large address spaces that are constructed by mapping files from the file system into virtual memory. Paging data in and out of the file system is likely to occur frequently. This requires a file system providing higher bandwidth than the original 512 byte UNIX one which provides only about two percent of the maximum disk bandwidth or about 20 kilobytes per second per arm [White80], [Smith81b].

Modifications have been made to the UNIX file system to improve its performance. Since the UNIX file system interface is well understood and not inherently slow, this development retained the abstraction and simply changed the underlying implementation to increase its throughput. Consequently users of the system have not been faced with massive software conversion.

Problems with file system performance have been dealt with extensively in the literature; see [Smith81a] for a survey. The UNIX operating system drew many of its ideas from Multics, a large, high performance operating system [Feiertag71]. Other work includes Hydra [Almes78], Spice [Thompson80], and a file system for a lisp environment [Symbolics81a].

A major goal of this project has been to build a file system that is extensible into a networked environment [Holler73]. Other work on network file systems describe centralized file servers [Accetta80], distributed file servers [Dion80], [Luniewski77], [Porcar82], and protocols to reduce the amount of information that must be transferred across a network [Symbolics81b], [Sturgis80].

---

† DEC, PDP, VAX, MASSBUS, and UNIBUS are trademarks of Digital Equipment Corporation.

2. Old File System

In the old file system developed at Bell Laboratories each disk drive contains one or more file systems.† A file system is described by its super-block, which contains the basic parameters of the file system. These include the number of data blocks in the file system, a count of the maximum number of files, and a pointer to a list of free blocks. All the free blocks in the system are chained together in a linked list. Within the file system are files. Certain files are distinguished as directories and contain pointers to files that may themselves be directories. Every file has a descriptor associated with it called an *inode*. The inode contains information describing ownership of the file, time stamps marking last modification and access times for the file, and an array of indices that point to the data blocks for the file. For the purposes of this section, we assume that the first 8 blocks of the file are directly referenced by values stored in the inode structure itself*. The inode structure may also contain references to indirect blocks containing further data block indices. In a file system with a 512 byte block size, a singly indirect block contains 128 further block addresses, a doubly indirect block contains 128 addresses of further single indirect blocks, and a triply indirect block contains 128 addresses of further doubly indirect blocks.

A traditional 150 megabyte UNIX file system consists of 4 megabytes of inodes followed by 146 megabytes of data. This organization segregates the inode information from the data; thus accessing a file normally incurs a long seek from its inode to its data. Files in a single directory are not typically allocated slots in consecutive locations in the 4 megabytes of inodes, causing many non-consecutive blocks to be accessed when executing operations on all the files in a directory.

The allocation of data blocks to files is also suboptimum. The traditional file system never transfers more than 512 bytes per disk transaction and often finds that the next sequential data block is not on the same cylinder, forcing seeks between 512 byte transfers. The combination of the small block size, limited read-ahead in the system, and many seeks severely limits file system throughput.

The first work at Berkeley on the UNIX file system attempted to improve both reliability and throughput. The reliability was improved by changing the file system so that all modifications of critical information were staged so that they could either be completed or repaired cleanly by a program after a crash [Kowalski78]. The file system performance was improved by a factor of more than two by changing the basic block size from 512 to 1024 bytes. The increase was because of two factors; each disk transfer accessed twice as much data, and most files could be described without need to access through any indirect blocks since the direct blocks contained twice as much data. The file system with these changes will henceforth be referred to as the *old file system*.

This performance improvement gave a strong indication that increasing the block size was a good method for improving throughput. Although the throughput had doubled, the old file system was still using only about four percent of the disk bandwidth. The main problem was that although the free list was initially ordered for optimal access, it quickly became scrambled as files were created and removed. Eventually the free list became entirely random causing files to have their blocks allocated randomly over the disk. This forced the disk to seek before every block access. Although old file systems provided transfer rates of up to 175 kilobytes per second when they were first created, this rate deteriorated to 30 kilobytes per second after a few weeks of moderate use because of randomization of their free block list. There was no way of restoring the performance an old file system except to dump, rebuild, and restore the file system. Another possibility would be to have a process that periodically reorganized the data on the disk to restore locality as suggested by [Maruyama76].

---

† A file system always resides on a single drive.
* The actual number may vary from system to system, but is usually in the range 5-13.

3. New file system organization

As in the old file system organization each disk drive contains one or more file systems. A file system is described by its super-block, that is located at the beginning of its disk partition. Because the super-block contains critical data it is replicated to protect against catastrophic loss. This is done at the time that the file system is created; since the super-block data does not change, the copies need not be referenced unless a head crash or other hard disk error causes the default super-block to be unusable.

To insure that it is possible to create files as large as 2↑32 bytes with only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of two greater than or equal to 4096. The block size of the file system is maintained in the super-block so it is possible for file systems with different block sizes to be accessible simultaneously on the same system. The block size must be decided at the time that the file system is created; it cannot be subsequently changed without rebuilding the file system.

The new file system organization partitions the disk into one or more areas called *cylinder groups*. A cylinder group is comprised of one or more consecutive cylinders on a disk. Associated with each cylinder group is some bookkeeping information that includes a redundant copy of the super-block, space for inodes, a bit map describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. For each cylinder group a static number of inodes is allocated at file system creation time. The current policy is to allocate one inode for each 2048 bytes of disk space, expecting this to be far more than will ever be needed.

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all the redundant information would be on the top platter. Thus a single hardware failure that destroyed the top platter could cause the loss of all copies of the redundant super-blocks. Thus the cylinder group bookkeeping information begins at a floating offset from the beginning of the cylinder group. The offset for each successive cylinder group is calculated to be about one track further from the beginning of the cylinder group. In this way the redundant information spirals down into the pack so that any single track, cylinder, or platter can be lost without losing all copies of the super-blocks. Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information is used for data blocks.†

3.1. Optimizing storage utilization

Data is laid out so that larger blocks can be transferred in a single disk transfer, greatly increasing file system throughput. As an example, consider a file in the new file system composed of 4096 byte data blocks. In the old file system this file would be composed of 1024 byte blocks. By increasing the block size, disk accesses in the new file system may transfer up to four times as much information per disk transaction. In large files, several 4096 byte blocks may be allocated from the same cylinder so that even larger data transfers are possible before initiating a seek.

The main problem with bigger blocks is that most UNIX file systems are composed of many small files. A uniformly large block size wastes space. Table 1 shows the effect of file system block size on the amount of wasted space in the file system. The machine measured to obtain these figures is one of our time sharing systems that has roughly 1.2 Gigabyte of on-line storage. The measurements are based on the active user file systems containing about 920 megabytes of formated space. The space wasted is measured as the percentage of space on the disk not containing user data. As the block size on the disk increases, the waste rises quickly, to an intolerable 45.6% waste with 4096 byte file system blocks.

---

† While it appears that the first cylinder group could be laid out with its super-block at the "known" location, this would not work for file systems with blocks sizes of 16K or greater, because of the requirement that the cylinder group information must begin at a block boundary.

Space used	% waste	Organization
775.2 Mb	0.0	Data only, no separation between files
807.8 Mb	4.2	Data only, each file starts on 512 byte boundary
828.7 Mb	6.9	512 byte block UNIX file system
866.5 Mb	11.8	1024 byte block UNIX file system
948.5 Mb	22.4	2048 byte block UNIX file system
1128.3 Mb	45.6	4096 byte block UNIX file system

Table 1 — Amount of wasted space as a function of block size.

To be able to use large blocks without undue waste, small files must be stored in a more efficient way. The new file system accomplishes this goal by allowing the division of a single file system block into one or more *fragments*. The file system fragment size is specified at the time that the file system is created; each file system block can be optionally broken into 2, 4, or 8 fragments, each of which is addressable. The lower bound on the size of these fragments is constrained by the disk sector size, typically 512 bytes. The block map associated with each cylinder group records the space availability at the fragment level; to determine block availability, aligned fragments are examined. Figure 1 shows a piece of a map from a 4096/1024 file system.

Bits in map	XXXX	XXOO	OOXX	OOOO
Fragment numbers	0-3	4-7	8-11	12-15
Block numbers	0	1	2	3

Figure 1 — Example layout of blocks and fragments in a 4096/1024 file system.

Each bit in the map records the status of a fragment; an "X" shows that the fragment is in use, while a "O" shows that the fragment is available for allocation. In this example, fragments 0−5, 10, and 11 are in use, while fragments 6−9, and 12−15 are free. Fragments of adjoining blocks cannot be used as a block, even if they are large enough. In this example, fragments 6−9 cannot be coalesced into a block; only fragments 12−15 are available for allocation as a block.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096 byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remainder of the block is made available for allocation to other files. As an example consider an 11000 byte file stored on a 4096/1024 byte file system. This file would uses two full size blocks and a 3072 byte fragment. If no 3072 byte fragments are available at the time the file is created, a full size block is split yielding the necessary 3072 byte fragment and an unused 1024 byte fragment. This remaining fragment can be allocated to another file as needed.

The granularity of allocation is the *write* system call. Each time data is written to a file, the system checks to see if the size of the file has increased*. If the file needs to hold the new data, one of three conditions exists:

1) There is enough space left in an already allocated block to hold the new data. The new data is written into the available space in the block.

2) Nothing has been allocated. If the new data contains more than 4096 bytes, a 4096 byte block is allocated and the first 4096 bytes of new data is written there. This process is repeated until less than 4096 bytes of new data remain. If the remaining new data to be written will fit in three or fewer 1024 byte pieces, an unallocated fragment is located, otherwise a 4096 byte block is located. The new data is written into the located piece.

---

* A program may be overwriting data in the middle of an existing file in which case space will already be allocated.

3)   A fragment has been allocated. If the number of bytes in the new data plus the number of bytes already in the fragment exceeds 4096 bytes, a 4096 byte block is allocated. The contents of the fragment is copied to the beginning of the block and the remainder of the block is filled with the new data. The process then continues as in (2) above. If the number of bytes in the new data plus the number of bytes already in the fragment will fit in three or fewer 1024 byte pieces, an unallocated fragment is located, otherwise a 4096 byte block is located. The contents of the previous fragment appended with the new data is written into the allocated piece.

The problem with allowing only a single fragment on a 4096/1024 byte file system is that data may be potentially copied up to three times as its requirements grow from a 1024 byte fragment to a 2048 byte fragment, then a 3072 byte fragment, and finally a 4096 byte block. The fragment reallocation can be avoided if the user program writes a full block at a time, except for a partial block at the end of the file. Because file systems with different block sizes may coexist on the same system, the file system interface been extended to provide the ability to determine the optimal size for a read or write. For files the optimal size is the block size of the file system on which the file is being accessed. For other objects, such as pipes and sockets, the optimal size is the underlying buffer size. This feature is used by the Standard Input/Output Library, a package used by most user programs. This feature is also used by certain system utilities such as archivers and loaders that do their own input and output management and need the highest possible file system bandwidth.

The space overhead in the 4096/1024 byte new file system organization is empirically observed to be about the same as in the 1024 byte old file system organization. A file system with 4096 byte blocks and 512 byte fragments has about the same amount of space overhead as the 512 byte block UNIX file system. The new file system is more space efficient than the 512 byte or 1024 byte file systems in that it uses the same amount of space for small files while requiring less indexing information for large files. This savings is offset by the need to use more space for keeping track of available free blocks. The net result is about the same disk utilization when the new file systems fragment size equals the old file systems block size.

In order for the layout policies to be effective, the disk cannot be kept completely full. Each file system maintains a parameter that gives the minimum acceptable percentage of file system blocks that can be free. If the the number of free blocks drops below this level only the system administrator can continue to allocate blocks. The value of this parameter can be changed at any time, even when the file system is mounted and active. The transfer rates to be given in section 4 were measured on file systems kept less than 90% full. If the reserve of free blocks is set to zero, the file system throughput rate tends to be cut in half, because of the inability of the file system to localize the blocks in a file. If the performance is impaired because of overfilling, it may be restored by removing enough files to obtain 10% free space. Access speed for files created during periods of little free space can be restored by recreating them once enough space is available. The amount of free space maintained must be added to the percentage of waste when comparing the organizations given in Table 1. Thus, a site running the old 1024 byte UNIX file system wastes 11.8% of the space and one could expect to fit the same amount of data into a 4096/512 byte new file system with 5% free space, since a 512 byte old file system wasted 6.9% of the space.

3.2. File system parameterization

Except for the initial creation of the free list, the old file system ignores the parameters of the underlying hardware. It has no information about either the physical characteristics of the mass storage device, or the hardware that interacts with it. A goal of the new file system is to parameterize the processor capabilities and mass storage characteristics so that blocks can be allocated in an optimum configuration dependent way. Parameters used include the speed of the processor, the hardware support for mass storage transfers, and the characteristics of the mass storage devices. Disk technology is constantly improving and a given installation can have several different disk technologies running on a single processor. Each file system is

parameterized so that it can adapt to the characteristics of the disk on which it is placed.

For mass storage devices such as disks, the new file system tries to allocate new blocks on the same cylinder as the previous block in the same file. Optimally, these new blocks will also be well positioned rotationally. The distance between "rotationally optimal" blocks varies greatly; it can be a consecutive block or a rotationally delayed block depending on system characteristics. On a processor with a channel that does not require any processor intervention between mass storage transfer requests, two consecutive disk blocks often can be accessed without suffering lost time because of an intervening disk revolution. For processors without such channels, the main processor must field an interrupt and prepare for a new disk transfer. The expected time to service this interrupt and schedule a new disk transfer depends on the speed of the main processor.

The physical characteristics of each disk include the number of blocks per track and the rate at which the disk spins. The allocation policy routines use this information to calculate the number of milliseconds required to skip over a block. The characteristics of the processor include the expected time to schedule an interrupt. Given the previous block allocated to a file, the allocation routines calculate the number of blocks to skip over so that the next block in a file will be coming into position under the disk head in the expected amount of time that it takes to start a new disk transfer operation. For programs that sequentially access large amounts of data, this strategy minimizes the amount of time spent waiting for the disk to position itself.

To ease the calculation of finding rotationally optimal blocks, the cylinder group summary information includes a count of the availability of blocks at different rotational positions. Eight rotational positions are distinguished, so the resolution of the summary information is 2 milliseconds for a typical 3600 revolution per minute drive.

The parameter that defines the minimum number of milliseconds between the completion of a data transfer and the initiation of another data transfer on the same cylinder can be changed at any time, even when the file system is mounted and active. If a file system is parameterized to lay out blocks with rotational separation of 2 milliseconds, and the disk pack is then moved to a system that has a processor requiring 4 milliseconds to schedule a disk operation, the throughput will drop precipitously because of lost disk revolutions on nearly every block. If the eventual target machine is known, the file system can be parameterized for it even though it is initially created on a different processor. Even if the move is not known in advance, the rotational layout delay can be reconfigured after the disk is moved so that all further allocation is done based on the characteristics of the new host.

3.3. Layout policies

The file system policies are divided into two distinct parts. At the top level are global policies that use file system wide summary information to make decisions regarding the placement of new inodes and data blocks. These routines are responsible for deciding the placement of new directories and files. They also calculate rotationally optimal block layouts, and decide when to force a long seek to a new cylinder group because there are insufficient blocks left in the current cylinder group to do reasonable layouts. Below the global policy routines are the local allocation routines that use a locally optimal scheme to lay out data blocks.

Two methods for improving file system performance are to increase the locality of reference to minimize seek latency as described by [Trivedi80], and to improve the layout of data to make larger transfers possible as described by [Nevalainen77]. The global layout policies try to improve performance by clustering related information. They cannot attempt to localize all data references, but must also try to spread unrelated data among different cylinder groups. If too much localization is attempted, the local cylinder group may run out of space forcing the data to be scattered to non-local cylinder groups. Taken to an extreme, total localization can result in a single huge cluster of data resembling the old file system. The global policies try to balance the two conflicting goals of localizing data that is concurrently accessed while spreading out unrelated data.

One allocatable resource is inodes. Inodes are used to describe both files and directories. Files in a directory are frequently accessed together. For example the "list directory" command often accesses the inode for each file in a directory. The layout policy tries to place all the files in a directory in the same cylinder group. To ensure that files are allocated throughout the disk, a different policy is used for directory allocation. A new directory is placed in the cylinder group that has a greater than average number of free inodes, and the fewest number of directories in it already. The intent of this policy is to allow the file clustering policy to succeed most of the time. The allocation of inodes within a cylinder group is done using a next free strategy. Although this allocates the inodes randomly within a cylinder group, all the inodes for each cylinder group can be read with 4 to 8 disk transfers. This puts a small and constant upper bound on the number of disk transfers required to access all the inodes for all the files in a directory as compared to the old file system where typically, one disk transfer is needed to get the inode for each file in a directory.

The other major resource is the data blocks. Since data blocks for a file are typically accessed together, the policy routines try to place all the data blocks for a file in the same cylinder group, preferably rotationally optimally on the same cylinder. The problem with allocating all the data blocks in the same cylinder group is that large files will quickly use up available space in the cylinder group, forcing a spill over to other areas. Using up all the space in a cylinder group has the added drawback that future allocations for any file in the cylinder group will also spill to other areas. Ideally none of the cylinder groups should ever become completely full. The solution devised is to redirect block allocation to a newly chosen cylinder group when a file exceeds 32 kilobytes, and at every megabyte thereafter. The newly chosen cylinder group is selected from those cylinder groups that have a greater than average number of free blocks left. Although big files tend to be spread out over the disk, a megabyte of data is typically accessible before a long seek must be performed, and the cost of one long seek per megabyte is small.

The global policy routines call local allocation routines with requests for specific blocks. The local allocation routines will always allocate the requested block if it is free. If the requested block is not available, the allocator allocates a free block of the requested size that is rotationally closest to the requested block. If the global layout policies had complete information, they could always request unused blocks and the allocation routines would be reduced to simple bookkeeping. However, maintaining complete information is costly; thus the implementation of the global layout policy uses heuristic guesses based on partial information.

If a requested block is not available the local allocator uses a four level allocation strategy:

1)   Use the available block rotationally closest to the requested block on the same cylinder.

2)   If there are no blocks available on the same cylinder, use a block within the same cylinder group.

3)   If the cylinder group is entirely full, quadratically rehash among the cylinder groups looking for a free block.

4)   Finally if the rehash fails, apply an exhaustive search.

The use of quadratic rehash is prompted by studies of symbol table strategies used in programming languages. File systems that are parameterized to maintain at least 10% free space almost never use this strategy; file systems that are run without maintaining any free space typically have so few free blocks that almost any allocation is random. Consequently the most important characteristic of the strategy used when the file system is low on space is that it be fast.

4. Performance

Ultimately, the proof of the effectiveness of the algorithms described in the previous section is the long term performance of the new file system.

Our empiric studies have shown that the inode layout policy has been effective. When running the "list directory" command on a large directory that itself contains many directories, the number of disk accesses for inodes is cut by a factor of two. The improvements are even more dramatic for large directories containing only files, disk accesses for inodes being cut by a factor of eight. This is most encouraging for programs such as spooling daemons that access many small files, since these programs tend to flood the disk request queue on the old file system.

Table 2 summarizes the measured throughput of the new file system. Several comments need to be made about the conditions under which these tests were run. The test programs measure the rate that user programs can transfer data to or from a file without performing any processing on it. These programs must write enough data to insure that buffering in the operating system does not affect the results. They should also be run at least three times in succession; the first to get the system into a known state and the second two to insure that the experiment has stabilized and is repeatable. The methodology and test results are discussed in detail in [Kridle83]†. The systems were running multi-user but were otherwise quiescent. There was no contention for either the cpu or the disk arm. The only difference between the UNIBUS and MASSBUS tests was the controller. All tests used an Ampex Capricorn 330 Megabyte Winchester disk. As Table 2 shows, all file system test runs were on a VAX 11/750. All file systems had been in production use for at least a month before being measured.

Type of File System	Processor and Bus Measured	Read		
		Speed	Bandwidth	% CPU
old 1024	750/UNIBUS	29 Kbytes/sec	29/1100 3%	11%
new 4096/1024	750/UNIBUS	221 Kbytes/sec	221/1100 20%	43%
new 8192/1024	750/UNIBUS	233 Kbytes/sec	233/1100 21%	29%
new 4096/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	73%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	54%

Table 2a — Reading rates of the old and new UNIX file systems.

Type of File System	Processor and Bus Measured	Write		
		Speed	Bandwidth	% CPU
old 1024	750/UNIBUS	48 Kbytes/sec	48/1100 4%	29%
new 4096/1024	750/UNIBUS	142 Kbytes/sec	142/1100 13%	43%
new 8192/1024	750/UNIBUS	215 Kbytes/sec	215/1100 19%	46%
new 4096/1024	750/MASSBUS	323 Kbytes/sec	323/1200 27%	94%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	95%

Table 2b — Writing rates of the old and new UNIX file systems.

Unlike the old file system, the transfer rates for the new file system do not appear to change over time. The throughput rate is tied much more strongly to the amount of free space that is maintained. The measurements in Table 2 were based on a file system run with 10% free space. Synthetic work loads suggest the performance deteriorates to about half the throughput rates given in Table 2 when no free space is maintained.

The percentage of bandwidth given in Table 2 is a measure of the effective utilization of the disk by the file system. An upper bound on the transfer rate from the disk is measured by doing 65536* byte reads from contiguous tracks on the disk. The bandwidth is calculated by

---

† A UNIX command that is similar to the reading test that we used is, "cp file /dev/null", where "file" is eight Megabytes long.
* This number, 65536, is the maximal I/O size supported by the VAX hardware; it is a remnant of the

comparing the data rates the file system is able to achieve as a percentage of this rate. Using this metric, the old file system is only able to use about 3-4% of the disk bandwidth, while the new file system uses up to 39% of the bandwidth.

In the new file system, the reading rate is always at least as fast as the writing rate. This is to be expected since the kernel must do more work when allocating blocks than when simply reading them. Note that the write rates are about the same as the read rates in the 8192 byte block file system; the write rates are slower than the read rates in the 4096 byte block file system. The slower write rates occur because the kernel has to do twice as many disk allocations per second, and the processor is unable to keep up with the disk transfer rate.

In contrast the old file system is about 50% faster at writing files than reading them. This is because the *write* system call is asynchronous and the kernel can generate disk transfer requests much faster than they can be serviced, hence disk transfers build up in the disk buffer cache. Because the disk buffer cache is sorted by minimum seek order, the average seek between the scheduled disk writes is much less than they would be if the data blocks are written out in the order in which they are generated. However when the file is read, the *read* system call is processed synchronously so the disk blocks must be retrieved from the disk in the order in which they are allocated. This forces the disk scheduler to do long seeks resulting in a lower throughput rate.

The performance of the new file system is currently limited by a memory to memory copy operation because it transfers data from the disk into buffers in the kernel address space and then spends 40% of the processor cycles copying these buffers to user address space. If the buffers in both address spaces are properly aligned, this transfer can be affected without copying by using the VAX virtual memory management hardware. This is especially desirable when large amounts of data are to be transferred. We did not implement this because it would change the semantics of the file system in two major ways; user programs would be required to allocate buffers on page boundaries, and data would disappear from buffers after being written.

Greater disk throughput could be achieved by rewriting the disk drivers to chain together kernel buffers. This would allow files to be allocated to contiguous disk blocks that could be read in a single disk transaction. Most disks contain either 32 or 48 512 byte sectors per track. The inability to use contiguous disk blocks effectively limits the performance on these disks to less than fifty percent of the available bandwidth. Since each track has a multiple of sixteen sectors it holds exactly two or three 8192 byte file system blocks, or four or six 4096 byte file system blocks. If the the next block for a file cannot be laid out contiguously, then the minimum spacing to the next allocatable block on any platter is between a sixth and a half a revolution. The implication of this is that the best possible layout without contiguous blocks uses only half of the bandwidth of any given track. If each track contains an odd number of sectors, then it is possible to resolve the rotational delay to any number of sectors by finding a block that begins at the desired rotational position on another track. The reason that block chaining has not been implemented is because it would require rewriting all the disk drivers in the system, and the current throughput rates are already limited by the speed of the available processors.

Currently only one block is allocated to a file at a time. A technique used by the DEMOS file system when it finds that a file is growing rapidly, is to preallocate several blocks at once, releasing them when the file is closed if they remain unused. By batching up the allocation the system can reduce the overhead of allocating at each write, and it can cut down on the number of disk writes needed to keep the block pointers on the disk synchronized with the block allocation [Powell79].

---

system's PDP-11 ancestry.

5. File system functional enhancements

The speed enhancements to the UNIX file system did not require any changes to the semantics or data structures viewed by the users. However several changes have been generally desired for some time but have not been introduced because they would require users to dump and restore all their file systems. Since the new file system already requires that all existing file systems be dumped and restored, these functional enhancements have been introduced at this time.

5.1. Long file names

File names can now be of nearly arbitrary length. The only user programs affected by this change are those that access directories. To maintain portability among UNIX systems that are not running the new file system, a set of directory access routines have been introduced that provide a uniform interface to directories on both old and new systems.

Directories are allocated in units of 512 bytes. This size is chosen so that each allocation can be transferred to disk in a single atomic operation. Each allocation unit contains variable-length directory entries. Each entry is wholly contained in a single allocation unit. The first three fields of a directory entry are fixed and contain an inode number, the length of the entry, and the length of the name contained in the entry. Following this fixed size information is the null terminated name, padded to a 4 byte boundary. The maximum length of a name in a directory is currently 255 characters.

Free space in a directory is held by entries that have a record length that exceeds the space required by the directory entry itself. All the bytes in a directory unit are claimed by the directory entries. This normally results in the last entry in a directory being large. When entries are deleted from a directory, the space is returned to the previous entry in the same directory unit by increasing its length. If the first entry of a directory unit is free, then its inode number is set to zero to show that it is unallocated.

5.2. File locking

The old file system had no provision for locking files. Processes that needed to synchronize the updates of a file had to create a separate "lock" file to synchronize their updates. A process would try to create a "lock" file. If the creation succeeded, then it could proceed with its update; if the creation failed, then it would wait, and try again. This mechanism had three drawbacks. Processes consumed CPU time, by looping over attempts to create locks. Locks were left lying around following system crashes and had to be cleaned up by hand. Finally, processes running as system administrator are always permitted to create files, so they had to use a different mechanism. While it is possible to get around all these problems, the solutions are not straight-forward, so a mechanism for locking files has been added.

The most general schemes allow processes to concurrently update a file. Several of these techniques are discussed in [Peterson83]. A simpler technique is to simply serialize access with locks. To attain reasonable efficiency, certain applications require the ability to lock pieces of a file. Locking down to the byte level has been implemented in the Onyx file system by [Bass81]. However, for the applications that currently run on the system, a mechanism that locks at the granularity of a file is sufficient.

Locking schemes fall into two classes, those using hard locks and those using advisory locks. The primary difference between advisory locks and hard locks is the decision of when to override them. A hard lock is always enforced whenever a program tries to access a file; an advisory lock is only applied when it is requested by a program. Thus advisory locks are only effective when all programs accessing a file use the locking scheme. With hard locks there must be some override policy implemented in the kernel, with advisory locks the policy is implemented by the user programs. In the UNIX system, programs with system administrator privilege can override any protection scheme. Because many of the programs that need to use locks run as system administrators, we chose to implement advisory locks rather than create a

protection scheme that was contrary to the UNIX philosophy or could not be used by system administration programs.

The file locking facilities allow cooperating programs to apply advisory *shared* or *exclusive* locks on files. Only one process has an exclusive lock on a file while multiple shared locks may be present. Both shared and exclusive locks cannot be present on a file at the same time. If any lock is requested when another process holds an exclusive lock, or an exclusive lock is requested when another process holds any lock, the open will block until the lock can be gained. Because shared and exclusive locks are advisory only, even if a process has obtained a lock on a file, another process can override the lock by opening the same file without a lock.

Locks can be applied or removed on open files, so that locks can be manipulated without needing to close and reopen the file. This is useful, for example, when a process wishes to open a file with a shared lock to read some information, to determine whether an update is required. It can then get an exclusive lock so that it can do a read, modify, and write to update the file in a consistent manner.

A request for a lock will cause the process to block if the lock can not be immediately obtained. In certain instances this is unsatisfactory. For example, a process that wants only to check if a lock is present would require a separate mechanism to find out this information. Consequently, a process may specify that its locking request should return with an error if a lock can not be immediately obtained. Being able to poll for a lock is useful to "daemon" processes that wish to service a spooling area. If the first instance of the daemon locks the directory where spooling takes place, later daemon processes can easily check to see if an active daemon exists. Since the lock is removed when the process exits or the system crashes, there is no problem with unintentional locks files that must be cleared by hand.

Almost no deadlock detection is attempted. The only deadlock detection made by the system is that the file descriptor to which a lock is applied does not currently have a lock of the same type (i.e. the second of two successive calls to apply a lock of the same type will fail). Thus a process can deadlock itself by requesting locks on two separate file descriptors for the same object.

5.3. Symbolic links

The 512 byte UNIX file system allows multiple directory entries in the same file system to reference a single file. The link concept is fundamental; files do not live in directories, but exist separately and are referenced by links. When all the links are removed, the file is deallocated. This style of links does not allow references across physical file systems, nor does it support inter-machine linkage. To avoid these limitations *symbolic links* have been added similar to the scheme used by *Multics* [Feiertag71].

A symbolic link is implemented as a file that contains a pathname. When the system encounters a symbolic link while interpreting a component of a pathname, the contents of the symbolic link is prepended to the rest of the pathname, and this name is interpreted to yield the resulting pathname. If the symbolic link contains an absolute pathname, the absolute pathname is used, otherwise the contents of the symbolic link is evaluated relative to the location of the link in the file hierarchy.

Normally programs do not want to be aware that there is a symbolic link in a pathname that they are using. However certain system utilities must be able to detect and manipulate symbolic links. Three new system calls provide the ability to detect, read, and write symbolic links, and seven system utilities were modified to use these calls.

In future Berkeley software distributions it will be possible to mount file systems from other machines within a local file system. When this occurs, it will be possible to create symbolic links that span machines.

5.4. Rename

Programs that create new versions of data files typically create the new version as a temporary file and then rename the temporary file with the original name of the data file. In the old UNIX file systems the renaming required three calls to the system. If the program were interrupted or the system crashed between these calls, the data file could be left with only its temporary name. To eliminate this possibility a single system call has been added that performs the rename in an atomic fashion to guarantee the existence of the original name.

In addition, the rename facility allows directories to be moved around in the directory tree hierarchy. The rename system call performs special validation checks to insure that the directory tree structure is not corrupted by the creation of loops or inaccessible directories. Such corruption would occur if a parent directory were moved into one of its descendants. The validation check requires tracing the ancestry of the target directory to insure that it does not include the directory being moved.

5.5. Quotas

The UNIX system has traditionally attempted to share all available resources to the greatest extent possible. Thus any single user can allocate all the available space in the file system. In certain environments this is unacceptable. Consequently, a quota mechanism has been added for restricting the amount of file system resources that a user can obtain. The quota mechanism sets limits on both the number of files and the number of disk blocks that a user may allocate. A separate quota can be set for each user on each file system. Each resource is given both a hard and a soft limit. When a program exceeds a soft limit, a warning is printed on the users terminal; the offending program is not terminated unless it exceeds its hard limit. The idea is that users should stay below their soft limit between login sessions, but they may use more space while they are actively working. To encourage this behavior, users are warned when logging in if they are over any of their soft limits. If they fail to correct the problem for too many login sessions, they are eventually reprimanded by having their soft limit enforced as their hard limit.

6. Software engineering

The preliminary design was done by Bill Joy in late 1980; he presented the design at The USENIX Conference held in San Francisco in January 1981. The implementation of his design was done by Kirk McKusick in the summer of 1981. Most of the new system calls were implemented by Sam Leffler. The code for enforcing quotas was implemented by Robert Elz at the University of Melbourne.

To understand how the project was done it is necessary to understand the interfaces that the UNIX system provides to the hardware mass storage systems. At the lowest level is a *raw disk*. This interface provides access to the disk as a linear array of sectors. Normally this interface is only used by programs that need to do disk to disk copies or that wish to dump file systems. However, user programs with proper access rights can also access this interface. A disk is usually formated with a file system that is interpreted by the UNIX system to provide a directory hierarchy and files. The UNIX system interprets and multiplexes requests from user programs to create, read, write, and delete files by allocating and freeing inodes and data blocks. The interpretation of the data on the disk could be done by the user programs themselves. The reason that it is done by the UNIX system is to synchronize the user requests, so that two processes do not attempt to allocate or modify the same resource simultaneously. It also allows access to be restricted at the file level rather than at the disk level and allows the common file system routines to be shared between processes.

The implementation of the new file system amounted to using a different scheme for formating and interpreting the disk. Since the synchronization and disk access routines themselves were not being changed, the changes to the file system could be developed by moving the file system interpretation routines out of the kernel and into a user program. Thus, the first step was to extract the file system code for the old file system from the UNIX kernel and change its requests to the disk driver to accesses to a raw disk. This produced a library of routines that mapped what would normally be system calls into read or write operations on the raw disk. This library was then debugged by linking it into the system utilities that copy, remove, archive, and restore files.

A new cross file system utility was written that copied files from the simulated file system to the one implemented by the kernel. This was accomplished by calling the simulation library to do a read, and then writing the resultant data by using the conventional write system call. A similar utility copied data from the kernel to the simulated file system by doing a conventional read system call and then writing the resultant data using the simulated file system library.

The second step was to rewrite the file system simulation library to interpret the new file system. By linking the new simulation library into the cross file system copying utility, it was possible to easily copy files from the old file system into the new one and from the new one to the old one. Having the file system interpretation implemented in user code had several major benefits. These included being able to use the standard system tools such as the debuggers to set breakpoints and single step through the code. When bugs were discovered, the offending problem could be fixed and tested without the need to reboot the machine. There was never a period where it was necessary to maintain two concurrent file systems in the kernel. Finally it was not necessary to dedicate a machine entirely to file system development, except for a brief period while the new file system was boot strapped.

The final step was to merge the new file system back into the UNIX kernel. This was done in less than two weeks, since the only bugs remaining were those that involved interfacing to the synchronization routines that could not be tested in the simulated system. Again the simulation system proved useful since it enabled files to be easily copied between old and new file systems regardless of which file system was running in the kernel. This greatly reduced the number of times that the system had to be rebooted.

The total design and debug time took about one man year. Most of the work was done on the file system utilities, and changing all the user programs to use the new facilities. The code changes in the kernel were minor, involving the addition of only about 800 lines of code

(including comments).

Acknowledgements

References

[Accetta80]   Accetta, M., Robertson, G., Satyanarayanan, M., and Thompson, M. "The Design of a Network-Based Central File System", Carnegie-Mellon University, Dept of Computer Science Tech Report, #CMU-CS-80-134

[Almes78]   Almes, G., and Robertson, G. "An Extensible File System for Hydra" Proceedings of the Third International Conference on Software Engineering, IEEE, May 1978.

[Bass81]   Bass, J. "Implementation Description for File Locking", Onyx Systems Inc, 73 E. Trimble Rd, San Jose, CA 95131 Jan 1981.

[Dion80]   Dion, J. "The Cambridge File Server", Operating Systems Review, 14, 4. Oct 1980. pp 26-35

[Eswaran74]   Eswaran, K. "Placement of records in a file and file allocation in a computer network", Proceedings IFIPS, 1974. pp 304-307

[Holler73]   Holler, J. "Files in Computer Networks", First European Workshop on Computer Networks, April 1973. pp 381-396

[Feiertag71]   Feiertag, R. J. and Organick, E. I., "The Multics Input-Output System", Proceedings of the Third Symposium on Operating Systems Principles, ACM, Oct 1971. pp 35-41

[Kridle83]   Kridle, R., and McKusick, M., "Performance Effects of Disk Subsystem Choices for VAX Systems Running 4.2BSD UNIX", Computer Systems Research Group, Dept of EECS, Berkeley, CA 94720, Technical Report #8.

[Kowalski78]   Kowalski, T. "FSCK - The UNIX System Check Program", Bell Laboratory, Murray Hill, NJ 07974. March 1978

[Luniewski77] Luniewski, A. "File Allocation in a Distributed System", MIT Laboratory for Computer Science, Dec 1977.

[Maruyama76]   Maruyama, K., and Smith, S. "Optimal reorganization of Distributed Space Disk Files", Communications of the ACM, 19, 11. Nov 1976. pp 634-642

[Nevalainen77] Nevalainen, O., Vesterinen, M. "Determining Blocking Factors for Sequential Files by Heuristic Methods", The Computer Journal, 20, 3. Aug 1977. pp 245-247

[Peterson83]   Peterson, G. "Concurrent Reading While Writing", ACM Transactions on Programming Languages and Systems, ACM, 5, 1. Jan 1983. pp 46-55

[Powell79]   Powell, M. "The DEMOS File System", Proceedings of the Sixth Symposium on Operating Systems Principles, ACM, Nov 1977. pp 33-42

[Porcar82]   Porcar, J. "File Migration in Distributed Computer Systems", Ph.D. Thesis, Lawrence Berkeley Laboratory Tech Report #LBL-14763.

[Ritchie74]   Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System", CACM 17, 7. July 1974. pp 365-375

[Smith81a]   Smith, A. "Input/Output Optimization and Disk Architectures: A Survey", Performance and Evaluation 1. Jan 1981. pp 104-117

[Smith81b]   Smith, A. "Bibliography on File and I/O System Optimization and Related Topics", Operating Systems Review, 15, 4. Oct 1981. pp 39-54

[Sturgis80]   Sturgis, H., Mitchell, J., and Israel, J. "Issues in the Design and Use of a Distributed File System", Operating Systems Review, 14, 3. pp 55-79

[Symbolics81a] "Symbolics File System", Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Aug 1981.

[Symbolics81b] "Chaosnet FILE Protocol". Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Sept 1981.

[Thompson79]   Thompson, K. "UNIX Implementation", Section 31, Volume 2B, UNIX Programmers Manual, Bell Laboratory, Murray Hill, NJ 07974. Jan 1979

[Thompson80]   Thompson, M. "Spice File System", Carnegie-Mellon University, Dept of Computer Science Tech Report, #CMU-CS-80-???

[Trivedi80]   Trivedi, K. "Optimal Selection of CPU Speed, Device Capabilities, and File Assignments", Journal of the ACM, 27, 3. July 1980. pp 457-473

[White80]   White, R. M. "Disk Storage Technology", Scientific American, 243(2), August 1980.

4.2BSD Networking Implementation Notes

Revised July, 1983

Samuel J. Leffler, William N. Joy, Robert S. Fabry

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

(415) 642-7780

ABSTRACT

This report describes the internal structure of the networking facilities developed for the 4.2BSD version of the UNIX* operating system for the VAX†. These facilities are based on several central abstractions which structure the external (user) view of network communication as well as the internal (system) implementation.

The report documents the internal structure of the networking system. The "4.2BSD System Manual" provides a description of the user interface to the networking facilities.

---

* UNIX is a trademark of Bell Laboratories.
† DEC, VAX, DECnet, and UNIBUS are trademarks of Digital Equipment Corporation.

TABLE OF CONTENTS

1. Introduction

This report describes the internal structure of facilities added to the 4.2BSD version of the UNIX operating system for the VAX. The system facilities provide a uniform user interface to networking within UNIX. In addition, the implementation introduces a structure for network communications which may be used by system implementors in adding new networking facilities. The internal structure is not visible to the user, rather it is intended to aid implementors of communication protocols and network services by providing a framework which promotes code sharing and minimizes implementation effort.

The reader is expected to be familiar with the C programming language and system interface, as described in the *4.2BSD System Manual* [Joy82a]. Basic understanding of network communication concepts is assumed; where required any additional ideas are introduced.

The remainder of this document provides a description of the system internals, avoiding, when possible, those portions which are utilized only by the interprocess communication facilities.

2. Overview

If we consider the International Standards Organization's (ISO) Open System Interconnection (OSI) model of network communication [ISO81] [Zimmermann80], the networking facilities described here correspond to a portion of the session layer (layer 3) and all of the transport and network layers (layers 2 and 1, respectively).

The network layer provides possibly imperfect data transport services with minimal addressing structure. Addressing at this level is normally host to host, with implicit or explicit routing optionally supported by the communicating agents.

At the transport layer the notions of reliable transfer, data sequencing, flow control, and service addressing are normally included. Reliability is usually managed by explicit acknowledgement of data delivered. Failure to acknowledge a transfer results in retransmission of the data. Sequencing may be handled by tagging each message handed to the network layer by a *sequence number* and maintaining state at the endpoints of communication to utilize received sequence numbers in reordering data which arrives out of order.

The session layer facilities may provide forms of addressing which are mapped into formats required by the transport layer, service authentication and client authentication, etc. Various systems also provide services such as data encryption and address and protocol translation.

The following sections begin by describing some of the common data structures and utility routines, then examine the internal layering. The contents of each layer and its interface are considered. Certain of the interfaces are protocol implementation specific. For these cases examples have been drawn from the Internet [Cerf78] protocol family. Later sections cover routing issues, the design of the raw socket interface and other miscellaneous topics.

3. Goals

The networking system was designed with the goal of supporting multiple *protocol families* and addressing styles. This required information to be "hidden" in common data structures which could be manipulated by all the pieces of the system, but which required interpretation only by the protocols which "controlled" it. The system described here attempts to minimize the use of shared data structures to those kept by a suite of protocols (a *protocol family*), and those used for rendezvous between "synchronous" and "asynchronous" portions of the system (e.g. queues of data packets are filled at interrupt time and emptied based on user requests).

A major goal of the system was to provide a framework within which new protocols and hardware could be easily be supported. To this end, a great deal of effort has been extended to create utility routines which hide many of the more complex and/or hardware dependent chores of networking. Later sections describe the utility routines and the underlying data structures they manipulate.

4. Internal address representation

Common to all portions of the system are two data structures. These structures are used to represent addresses and various data objects. Addresses, internally are described by the *sockaddr* structure,

```
struct sockaddr {
        short      sa_family;        /* data format identifier */
        char       sa_data[14];      /* address */
};
```

All addresses belong to one or more *address families* which define their format and interpretation. The *sa_family* field indicates which address family the address belongs to, the *sa_data* field contains the actual data value. The size of the data field, 14 bytes, was selected based on a study of current address formats*.

---

* Later versions of the system support variable length addresses.

5. Memory management

A single mechanism is used for data storage: memory buffers, or *mbuf*'s. An mbuf is a structure of the form:

```
struct mbuf {
        struct      mbuf *m_next;       /* next buffer in chain */
        u_long      m_off;              /* offset of data */
        short       m_len;              /* amount of data in this mbuf */
        short       m_type;             /* mbuf type (accounting) */
        u_char      m_dat[MLEN];        /* data storage */
        struct      mbuf *m_act;        /* link in higher-level mbuf list */
};
```

The *m_next* field is used to chain mbufs together on linked lists, while the *m_act* field allows lists of mbufs to be accumulated. By convention, the mbufs common to a single object (for example, a packet) are chained together with the *m_next* field, while groups of objects are linked via the *m_act* field (possibly when in a queue).

Each mbuf has a small data area for storing information, *m_dat*. The *m_len* field indicates the amount of data, while the *m_off* field is an offset to the beginning of the data from the base of the mbuf. Thus, for example, the macro *mtod*, which converts a pointer to an mbuf to a pointer to the data stored in the mbuf, has the form

```
#define  mtod(x,t)              ((t)((int)(x) + (x)->m_off))
```

(note the *t* parameter, a C type cast, is used to cast the resultant pointer for proper assignment).

In addition to storing data directly in the mbuf's data area, data of page size may be also be stored in a separate area of memory. The mbuf utility routines maintain a pool of pages for this purpose and manipulate a private page map for such pages. The virtual addresses of these data pages precede those of mbufs, so when pages of data are separated from an mbuf, the mbuf data offset is a negative value. An array of reference counts on pages is also maintained so that copies of pages may be made without core to core copying (copies are created simply by duplicating the relevant page table entries in the data page map and incrementing the associated reference counts for the pages). Separate data pages are currently used only when copying data from a user process into the kernel, and when bringing data in at the hardware level. Routines which manipulate mbufs are not normally aware if data is stored directly in the mbuf data array, or if it is kept in separate pages.

The following utility routines are available for manipulating mbuf chains:

m = m_copy(m0, off, len);
> The *m_copy* routine create a copy of all, or part, of a list of the mbufs in *m0*. *Len* bytes of data, starting *off* bytes from the front of the chain, are copied. Where possible, reference counts on pages are used instead of core to core copies. The original mbuf chain must have at least *off* + *len* bytes of data. If *len* is specified as M_COPYALL, all the data present, offset as before, is copied.

m_cat(m, n);
> The mbuf chain, *n*, is appended to the end of *m*. Where possible, compaction is performed.

m_adj(m, diff);
> The mbuf chain, *m* is adjusted in size by *diff* bytes. If *diff* is non-negative, *diff* bytes are shaved off the front of the mbuf chain. If *diff* is negative, the alteration is performed from back to front. No space is reclaimed in this operation, alterations are accomplished by changing the *m_len* and *m_off* fields of mbufs.

m = m_pullup(m0, size);
> After a successful call to *m_pullup*, the mbuf at the head of the returned list, *m*, is

guaranteed to have at least *size* bytes of data in contiguous memory (allowing access via a pointer, obtained using the *mtod* macro). If the original data was less than *size* bytes long, *len* was greater than the size of an mbuf data area (112 bytes), or required resources were unavailable, *m* is 0 and the original mbuf chain is deallocated.

This routine is particularly useful when verifying packet header lengths on reception. For example, if a packet is received and only 8 of the necessary 16 bytes required for a valid packet header are present at the head of the list of mbufs representing the packet, the remaining 8 bytes may be "pulled up" with a single *m_pullup* call. If the call fails the invalid packet will have been discarded.

By insuring mbufs always reside on 128 byte boundaries it is possible to always locate the mbuf associated with a data area by masking off the low bits of the virtual address. This allows modules to store data structures in mbufs and pass them around without concern for locating the original mbuf when it comes time to free the structure. The *dtom* macro is used to convert a pointer into an mbuf's data area to a pointer to the mbuf,

```
#define   dtom(x)   ((struct mbuf *)((int)x & ~(MSIZE-1)))
```

Mbufs are used for dynamically allocated data structures such as sockets, as well as memory allocated for packets. Statistics are maintained on mbuf usage and can be viewed by users using the *netstat*(1) program.

6. Internal layering

The internal structure of the network system is divided into three layers. These layers correspond to the services provided by the socket abstraction, those provided by the communication protocols, and those provided by the hardware interfaces. The communication protocols are normally layered into two or more individual cooperating layers, though they are collectively viewed in the system as one layer providing services supportive of the appropriate socket abstraction.

The following sections describe the properties of each layer in the system and the interfaces each must conform to.

6.1. Socket layer

The socket layer deals with the interprocess communications facilities provided by the system. A socket is a bidirectional endpoint of communication which is "typed" by the semantics of communication it supports. The system calls described in the *4.2BSD System Manual* are used to manipulate sockets.

A socket consists of the following data structure:

```
struct socket {
        short       so_type;            /* generic type */
        short       so_options;         /* from socket call */
        short       so_linger;          /* time to linger while closing */
        short       so_state;           /* internal state flags */
        caddr_t     so_pcb;             /* protocol control block */
        struct      protosw *so_proto;  /* protocol handle */
        struct      socket *so_head;    /* back pointer to accept socket */
        struct      socket *so_q0;      /* queue of partial connections */
        short       so_q0len;           /* partials on so_q0 */
        struct      socket *so_q;       /* queue of incoming connections */
        short       so_qlen;            /* number of connections on so_q */
        short       so_qlimit;          /* max number queued connections */
        struct      sockbuf so_snd;     /* send queue */
        struct      sockbuf so_rcv;     /* receive queue */
        short       so_timeo;           /* connection timeout */
        u_short     so_error;           /* error affecting connection */
        short       so_oobmark;         /* chars to oob mark */
        short       so_pgrp;            /* pgrp for signals */
};
```

Each socket contains two data queues, *so_rcv* and *so_snd*, and a pointer to routines which provide supporting services. The type of the socket, *so_type* is defined at socket creation time and used in selecting those services which are appropriate to support it. The supporting protocol is selected at socket creation time and recorded in the socket data structure for later use. Protocols are defined by a table of procedures, the *protosw* structure, which will be described in detail later. A pointer to a protocol specific data structure, the "protocol control block" is also present in the socket structure. Protocols control this data structure and it normally includes a back pointer to the parent socket structure(s) to allow easy lookup when returning information to a user (for example, placing an error number in the *so_error* field). The other entries in the socket structure are used in queueing connection requests, validating user requests, storing socket characteristics (e.g. options supplied at the time a socket is created), and maintaining a socket's state.

Processes "rendezvous at a socket" in many instances. For instance, when a process wishes to extract data from a socket's receive queue and it is empty, or lacks sufficient data to satisfy the request, the process blocks, supplying the address of the receive queue as an "wait channel' to be used in notification. When data arrives for the process and is placed in the

socket's queue, the blocked process is identified by the fact it is waiting "on the queue".

6.1.1. Socket state

A socket's state is defined from the following:

```
#define  SS_NOFDREF          0x001    /* no file table ref any more */
#define  SS_ISCONNECTED      0x002    /* socket connected to a peer */
#define  SS_ISCONNECTING     0x004    /* in process of connecting to peer */
#define  SS_ISDISCONNECTING  0x008    /* in process of disconnecting */
#define  SS_CANTSENDMORE     0x010    /* can't send more data to peer */
#define  SS_CANTRCVMORE      0x020    /* can't receive more data from peer */
#define  SS_CONNAWAITING     0x040    /* connections awaiting acceptance */
#define  SS_RCVATMARK        0x080    /* at mark on input */

#define  SS_PRIV             0x100    /* priviledged */
#define  SS_NBIO             0x200    /* non-blocking ops */
#define  SS_ASYNC            0x400    /* async i/o notify */
```

The state of a socket is manipulated both by the protocols and the user (through system calls). When a socket is created the state is defined based on the type of input/output the user wishes to perform. "Non-blocking" I/O implies a process should never be blocked to await resources. Instead, any call which would block returns prematurely with the error EWOULD-BLOCK (the service request may be partially fulfilled, e.g. a request for more data than is present).

If a process requested "asynchronous" notification of events related to the socket the SIGIO signal is posted to the process. An event is a change in the socket's state, examples of such occurances are: space becoming available in the send queue, new data available in the receive queue, connection establishment or disestablishment, etc.

A socket may be marked "priviledged" if it was created by the super-user. Only priviledged sockets may send broadcast packets, or bind addresses in priviledged portions of an address space.

6.1.2. Socket data queues

A socket's data queue contains a pointer to the data stored in the queue and other entries related to the management of the data. The following structure defines a data queue:

```
struct sockbuf {
        short    sb_cc;          /* actual chars in buffer */
        short    sb_hiwat;       /* max actual char count */
        short    sb_mbcnt;       /* chars of mbufs used */
        short    sb_mbmax;       /* max chars of mbufs to use */
        short    sb_lowat;       /* low water mark */
        short    sb_timeo;       /* timeout */
        struct   mbuf *sb_mb;    /* the mbuf chain */
        struct   proc *sb_sel;   /* process selecting read/write */
        short    sb_flags;       /* flags, see below */
};
```

Data is stored in a queue as a chain of mbufs. The actual count of characters as well as high and low water marks are used by the protocols in controlling the flow of data. The socket routines cooperate in implementing the flow control policy by blocking a process when it requests to send data and the high water mark has been reached, or when it requests to receive data and less than the low water mark is present (assuming non-blocking I/O has not been specified).

When a socket is created, the supporting protocol "reserves" space for the send and receive queues of the socket. The actual storage associated with a socket queue may fluctuate during a socket's lifetime, but is assumed this reservation will always allow a protocol to acquire enough memory to satisfy the high water marks.

The timeout and select values are manipulated by the socket routines in implementing various portions of the interprocess communications facilities and will not be described here.

A socket queue has a number of flags used in synchronizing access to the data and in acquiring resources;

```
#define  SB_LOCK        0x01   /* lock on data queue (so_rcv only) */
#define  SB_WANT        0x02   /* someone is waiting to lock */
#define  SB_WAIT        0x04   /* someone is waiting for data/space */
#define  SB_SEL         0x08   /* buffer is selected */
#define  SB_COLL        0x10   /* collision selecting */
```

The last two flags are manipulated by the system in implementing the select mechanism.

6.1.3. Socket connection queueing

In dealing with connection oriented sockets (e.g. SOCK_STREAM) the two sides are considered distinct. One side is termed *active*, and generates connection requests. The other side is called *passive* and accepts connection requests.

From the passive side, a socket is created with the option SO_ACCEPTCONN specified, creating two queues of sockets: *so_q0* for connections in progress and *so_q* for connections already made and awaiting user acceptance. As a protocol is preparing incoming connections, it creates a socket structure queued on *so_q0* by calling the routine *sonewconn()*. When the connection is established, the socket structure is then transfered to *so_q*, making it available for an accept.

If an SO_ACCEPTCONN socket is closed with sockets on either *so_q0* or *so_q*, these sockets are dropped.

6.2. Protocol layer(s)

Protocols are described by a set of entry points and certain socket visible characteristics, some of which are used in deciding which socket type(s) they may support.

An entry in the "protocol switch" table exists for each protocol module configured into the system. It has the following form:

```
struct protosw {
        short       pr_type;            /* socket type used for */
        short       pr_family;          /* protocol family */
        short       pr_protocol;        /* protocol number */
        short       pr_flags;           /* socket visible attributes */
/* protocol-protocol hooks */
        int         (*pr_input)();      /* input to protocol (from below) */
        int         (*pr_output)();     /* output to protocol (from above) */
        int         (*pr_ctlinput)();   /* control input (from below) */
        int         (*pr_ctloutput)();  /* control output (from above) */
/* user-protocol hook */
        int         (*pr_usrreq)();     /* user request */
/* utility hooks */
        int         (*pr_init)();       /* initialization routine */
        int         (*pr_fasttimo)();   /* fast timeout (200ms) */
        int         (*pr_slowtimo)();   /* slow timeout (500ms) */
        int         (*pr_drain)();      /* flush any excess space possible */
};
```

A protocol is called through the *pr_init* entry before any other. Thereafter it is called every 200 milliseconds through the *pr_fasttimo* entry and every 500 milliseconds through the *pr_slowtimo* for timer based actions. The system will call the *pr_drain* entry if it is low on space and this should throw away any non-critical data.

Protocols pass data between themselves as chains of mbufs using the *pr_input* and *pr_output* routines. *Pr_input* passes data up (towards the user) and *pr_output* passes it down (towards the network); control information passes up and down on *pr_ctlinput* and *pr_ctloutput*. The protocol is responsible for the space occupied by any the arguments to these entries and must dispose of it.

The *pr_userreq* routine interfaces protocols to the socket code and is described below.

The *pr_flags* field is constructed from the following values:

```
#define  PR_ATOMIC        0x01    /* exchange atomic messages only */
#define  PR_ADDR          0x02    /* addresses given with messages */
#define  PR_CONNREQUIRED  0x04    /* connection required by protocol */
#define  PR_WANTRCVD      0x08    /* want PRU_RCVD calls */
#define  PR_RIGHTS        0x10    /* passes capabilities */
```

Protocols which are connection-based specify the PR_CONNREQUIRED flag so that the socket routines will never attempt to send data before a connection has been established. If the PR_WANTRCVD flag is set, the socket routines will notfiy the protocol when the user has removed data from the socket's receive queue. This allows the protocol to implement acknowledgement on user receipt, and also update windowing information based on the amount of space available in the receive queue. The PR_ADDR field indicates any data placed in the socket's receive queue will be preceded by the address of the sender. The PR_ATOMIC flag specifies each *user* request to send data must be performed in a single *protocol* send request; it is the protocol's responsibility to maintain record boundaries on data to be sent. The PR_RIGHTS flag indicates the protocol supports the passing of capabilities; this is currently used only the protocols in the UNIX protocol family.

When a socket is created, the socket routines scan the protocol table looking for an appropriate protocol to support the type of socket being created. The *pr_type* field contains one of the possible socket types (e.g. SOCK_STREAM), while the *pr_family* field indicates which protocol family the protocol belongs to. The *pr_protocol* field contains the protocol number of the protocol, normally a well known value.

6.3. Network-interface layer

Each network-interface configured into a system defines a path through which packets may be sent and received. Normally a hardware device is associated with this interface, though there is no requirement for this (for example, all systems have a software "loopback" interface used for debugging and performance analysis). In addition to manipulating the hardware device, an interface module is responsible for encapsulation and deencapsulation of any low level header information required to deliver a message to it's destination. The selection of which interface to use in delivering packets is a routing decision carried out at a higher level than the network-interface layer. Each interface normally identifies itself at boot time to the routing module so that it may be selected for packet delivery.

An interface is defined by the following structure,

```
struct ifnet {
        char    *if_name;           /* name, e.g. "en" or "lo" */
        short   if_unit;            /* sub-unit for lower level driver */
        short   if_mtu;             /* maximum transmission unit */
        int     if_net;             /* network number of interface */
        short   if_flags;           /* up/down, broadcast, etc. */
        short   if_timer;           /* time 'til if_watchdog called */
        int     if_host[2];         /* local net host number */
        struct  sockaddr if_addr;   /* address of interface */
        union {
                struct          sockaddr ifu_broadaddr;
                struct          sockaddr ifu_dstaddr;
        } if_ifu;
        struct  ifqueue if_snd;     /* output queue */
        int     (*if_init)();       /* init routine */
        int     (*if_output)();     /* output routine */
        int     (*if_ioctl)();      /* ioctl routine */
        int     (*if_reset)();      /* bus reset routine */
        int     (*if_watchdog)();   /* timer routine */
        int     if_ipackets;        /* packets received on interface */
        int     if_ierrors;         /* input errors on interface */
        int     if_opackets;        /* packets sent on interface */
        int     if_oerrors;         /* output errors on interface */
        int     if_collisions;      /* collisions on csma interfaces */
        struct  ifnet *if_next;
};
```

Each interface has a send queue and routines used for initialization, *if_init*, and output, *if_output*. If the interface resides on a system bus, the routine *if_reset* will be called after a bus reset has been performed. An interface may also specify a timer routine, *if_watchdog*, which should be called every *if_timer* seconds (if non-zero).

The state of an interface and certain characteristics are stored in the *if_flags* field. The following values are possible:

```
#define IFF_UP          0x1     /* interface is up */
#define IFF_BROADCAST   0x2     /* broadcast address valid */
#define IFF_DEBUG       0x4     /* turn on debugging */
#define IFF_ROUTE       0x8     /* routing entry installed */
#define IFF_POINTOPOINT 0x10    /* interface is point-to-point link */
#define IFF_NOTRAILERS  0x20    /* avoid use of trailers */
#define IFF_RUNNING     0x40    /* resources allocated */
#define IFF_NOARP       0x80    /* no address resolution protocol */
```

If the interface is connected to a network which supports transmission of *broadcast* packets, the IFF_BROADCAST flag will be set and the *if_broadaddr* field will contain the address to be used in sending or accepting a broadcast packet. If the interface is associated with a point to point hardware link (for example, a DEC DMR-11), the IFF_POINTOPOINT flag will be set and *if_dstaddr* will contain the address of the host on the other side of the connection. These addresses and the local address of the interface, *if_addr*, are used in filtering incoming packets. The interface sets IFF_RUNNING after it has allocated system resources and posted an initial read on the device it manages. This state bit is used to avoid multiple allocation requests when an interface's address is changed. The IFF_NOTRAILERS flag indicates the interface should refrain from using a *trailer* encapsulation on outgoing packets; *trailer* protocols are described in section 14. The IFF_NOARP flag indicates the interface should not use an "address resolution protocol" in mapping internetwork addresses to local network addresses.

The information stored in an *ifnet* structure for point to point communication devices is not currently used by the system internally. Rather, it is used by the user level routing process in determining host network connections and in initially devising routes (refer to chapter 10 for more information).

Various statistics are also stored in the interface structure. These may be viewed by users using the *netstat*(1) program.

The interface address and flags may be set with the SIOCSIFADDR and SIOCSIFFLAGS ioctls. SIOCSIFADDR is used to initially define each interface's address; SIOGSIFFLAGS can be used to mark an interface down and perform site-specific configuration.

6.3.1. UNIBUS interfaces

All hardware related interfaces currently reside on the UNIBUS. Consequently a common set of utility routines for dealing with the UNIBUS has been developed. Each UNIBUS interface utilizes a structure of the following form:

```
struct   ifuba {
         short       ifu_uban;               /* uba number */
         short       ifu_hlen;               /* local net header length */
         struct      uba_regs *ifu_uba;      /* uba regs, in vm */
         struct ifrw {
                     caddr_t     ifrw_addr;  /* virt addr of header */
                     int         ifrw_bdp;   /* unibus bdp */
                     int         ifrw_info;  /* value from ubaalloc */
                     int         ifrw_proto; /* map register prototype */
                     struct      pte *ifrw_mr;/* base of map registers */
         } ifu_r, ifu_w;
         struct      pte ifu_wmap[IF_MAXNUBAMR];/* base pages for output */
         short       ifu_xswapd;             /* mask of clusters swapped */
         short       ifu_flags;              /* used during uballoc's */
         struct      mbuf *ifu_xtofree;      /* pages being dma'd out */
};
```

The *if_uba* structure describes UNIBUS resources held by an interface. IF_NUBAMR map registers are held for datagram data, starting at *ifr_mr*. UNIBUS map register *ifr_mr*[−1] maps the local network header ending on a page boundary. UNIBUS data paths are reserved for read and for write, given by *ifr_bdp*. The prototype of the map registers for read and for write is saved in *ifr_proto*.

When write transfers are not full pages on page boundaries the data is just copied into the pages mapped on the UNIBUS and the transfer is started. If a write transfer is of a (1024 byte) page size and on a page boundary, UNIBUS page table entries are swapped to reference the pages, and then the initial pages are remapped from *ifu_wmap* when the transfer completes.

When read transfers give whole pages of data to be input, page frames are allocated from a network page list and traded with the pages already containing the data, mapping the allocated pages to replace the input pages for the next UNIBUS data input.

The following utility routines are available for use in writing network interface drivers, all use the *ifuba* structure described above.

if_ubainit(ifu, uban, hlen, nmr);

> *if_ubainit* allocates resources on UNIBUS adaptor *uban* and stores the resultant information in the *ifuba* structure pointed to by *ifu*. It is called only at boot time or after a UNIBUS reset. Two data paths (buffered or unbuffered, depending on the *ifu_flags* field) are allocated, one for reading and one for writing. The *nmr* parameter indicates the number of UNIBUS mapping registers required to map a maximal sized packet onto the UNIBUS, while *hlen* specifies the size of a local network header, if any, which should be mapped separately from the data (see the description of trailer protocols in chapter 14). Sufficient UNIBUS mapping registers and pages of memory are allocated to initialize the input data path for an initial read. For the output data path, mapping registers and pages of memory are also allocated and mapped onto the UNIBUS. The pages associated with the output data path are held in reserve in the event a write requires copying non-page-aligned data (see *if_wubaput* below). If *if_ubainit* is called with resources already allocated, they will be used instead of allocating new ones (this normally occurs after a UNIBUS reset). A 1 is returned when allocation and initialization is successful, 0 otherwise.

m = if_rubaget(ifu, totlen, off0);

> *if_rubaget* pulls read data off an interface. *totlen* specifies the length of data to be obtained, not counting the local network header. If *off0* is non-zero, it indicates a byte offset to a trailing local network header which should be copied into a separate mbuf and prepended to the front of the resultant mbuf chain. When page sized units of data are present and are page-aligned, the previously mapped data pages are remapped into the mbufs and swapped with fresh pages; thus avoiding any copying. A 0 return value indicates a failure to allocate resources.

if_wubaput(ifu, m);

> *if_wubaput* maps a chain of mbufs onto a network interface in preparation for output. The chain includes any local network header, which is copied so that it resides in the mapped and aligned I/O space. Any other mbufs which contained non page sized data portions are also copied to the I/O space. Pages mapped from a previous output operation (no longer needed) are unmapped and returned to the network page pool.

7. Socket/protocol interface

The interface between the socket routines and the communication protocols is through the *pr_usrreq* routine defined in the protocol switch table. The following requests to a protocol module are possible:

```
#define   PRU_ATTACH        0     /* attach protocol */
#define   PRU_DETACH        1     /* detach protocol */
#define   PRU_BIND          2     /* bind socket to address */
#define   PRU_LISTEN        3     /* listen for connection */
#define   PRU_CONNECT       4     /* establish connection to peer */
#define   PRU_ACCEPT        5     /* accept connection from peer */
#define   PRU_DISCONNECT    6     /* disconnect from peer */
#define   PRU_SHUTDOWN      7     /* won't send any more data */
#define   PRU_RCVD          8     /* have taken data; more room now */
#define   PRU_SEND          9     /* send this data */
#define   PRU_ABORT         10    /* abort (fast DISCONNECT, DETATCH) */
#define   PRU_CONTROL       11    /* control operations on protocol */
#define   PRU_SENSE         12    /* return status into m */
#define   PRU_RCVOOB        13    /* retrieve out of band data */
#define   PRU_SENDOOB       14    /* send out of band data */
#define   PRU_SOCKADDR      15    /* fetch socket's address */
#define   PRU_PEERADDR      16    /* fetch peer's address */
#define   PRU_CONNECT2      17    /* connect two sockets */
/* begin for protocols internal use */
#define   PRU_FASTTIMO      18    /* 200ms timeout */
#define   PRU_SLOWTIMO      19    /* 500ms timeout */
#define   PRU_PROTORCV      20    /* receive from below */
#define   PRU_PROTOSEND     21    /* send to below */
```

A call on the user request routine is of the form,

```
error = (*protosw[].pr_usrreq)(up, req, m, addr, rights);
int error; struct socket *up; int req; struct mbuf *m, *rights; caddr_t addr;
```

The mbuf chain, *m*, and the address are optional parameters. The *rights* parameter is an optional pointer to an mbuf chain containing user specified capabilities (see the *sendmsg* and *recvmsg* system calls). The protocol is responsible for disposal of both mbuf chains. A non-zero return value gives a UNIX error number which should be passed to higher level software. The following paragraphs describe each of the requests possible.

PRU_ATTACH

When a protocol is bound to a socket (with the *socreate* system call) the protocol module is called with this request. It is the responsibility of the protocol module to allocate any resources necessary. The "attach" request will always precede any of the other requests, and should not occur more than once.

PRU_DETACH

This is the antithesis of the attach request, and is used at the time a socket is deleted. The protocol module may deallocate any resources assigned to the socket.

PRU_BIND

When a socket is initially created it has no address bound to it. This request indicates an address should be bound to an existing socket. The protocol module must verify the requested address is valid and available for use.

PRU_LISTEN

The "listen" request indicates the user wishes to listen for incoming connection requests on the associated socket. The protocol module should perform any state changes needed to carry out this request (if possible). A "listen" request always precedes a request to

accept a connection.

PRU_CONNECT

The "connect" request indicates the user wants to a establish an association. The *addr* parameter supplied describes the peer to be connected to. The effect of a connect request may vary depending on the protocol. Virtual circuit protocols, such as TCP [Postel80b], use this request to initiate establishment of a TCP connection. Datagram protocols, such as UDP [Postel79], simply record the peer's address in a private data structure and use it to tag all outgoing packets. There are no restrictions on how many times a connect request may be used after an attach. If a protocol supports the notion of *multi-casting*, it is possible to use multiple connects to establish a multi-cast group. Alternatively, an association may be broken by a PRU_DISCONNECT request, and a new association created with a subsequent connect request; all without destroying and creating a new socket.

PRU_ACCEPT

Following a successful PRU_LISTEN request and the arrival of one or more connections, this request is made to indicate the user has accepted the first connection on the queue of pending connections. The protocol module should fill in the supplied address buffer with the address of the connected party.

PRU_DISCONNECT

Eliminate an association created with a PRU_CONNECT request.

PRU_SHUTDOWN

This call is used to indicate no more data will be sent and/or received (the *addr* parameter indicates the direction of the shutdown, as encoded in the *soshutdown* system call). The protocol may, at its discretion, deallocate any data structures related to the shutdown.

PRU_RCVD

This request is made only if the protocol entry in the protocol switch table includes the PR_WANTRCVD flag. When a user removes data from the receive queue this request will be sent to the protocol module. It may be used to trigger acknowledgements, refresh windowing information, initiate data transfer, etc.

PRU_SEND

Each user request to send data is translated into one or more PRU_SEND requests (a protocol may indicate a single user send request must be translated into a single PRU_SEND request by specifying the PR_ATOMIC flag in its protocol description). The data to be sent is presented to the protocol as a list of mbufs and an address is, optionally, supplied in the *addr* parameter. The protocol is responsible for preserving the data in the socket's send queue if it is not able to send it immediately, or if it may need it at some later time (e.g. for retransmission).

PRU_ABORT

This request indicates an abnormal termination of service. The protocol should delete any existing association(s).

PRU_CONTROL

The "control" request is generated when a user performs a UNIX *ioctl* system call on a socket (and the ioctl is not intercepted by the socket routines). It allows protocol-specific operations to be provided outside the scope of the common socket interface. The *addr* parameter contains a pointer to a static kernel data area where relevant information may be obtained or returned. The *m* parameter contains the actual *ioctl* request code (note the non-standard calling convention).

PRU_SENSE

The "sense" request is generated when the user makes an *fstat* system call on a socket; it requests status of the associated socket. There currently is no common format for the status returned. Information which might be returned includes per-connection statistics, protocol state, resources currently in use by the connection, the optimal transfer size for the connection (based on windowing information and maximum packet size). The *addr*

parameter contains a pointer to a static kernel data area where the status buffer should be placed.

PRU_RCVOOB

Any "out-of-band" data presently available is to be returned. An mbuf is passed in to the protocol module and the protocol should either place data in the mbuf or attach new mbufs to the one supplied if there is insufficient space in the single mbuf.

PRU_SENDOOB

Like PRU_SEND, but for out-of-band data.

PRU_SOCKADDR

The local address of the socket is returned, if any is currently bound to the it. The address format (protocol specific) is returned in the *addr* parameter.

PRU_PEERADDR

The address of the peer to which the socket is connected is returned. The socket must be in a SS_ISCONNECTED state for this request to be made to the protocol. The address format (protocol specific) is returned in the *addr* parameter.

PRU_CONNECT2

The protocol module is supplied two sockets and requested to establish a connection between the two without binding any addresses, if possible. This call is used in implementing the system call.

The following requests are used internally by the protocol modules and are never generated by the socket routines. In certain instances, they are handed to the *pr_usrreq* routine solely for convenience in tracing a protocol's operation (e.g. PRU_SLOWTIMO).

PRU_FASTTIMO

A "fast timeout" has occured. This request is made when a timeout occurs in the protocol's *pr_fastimo* routine. The *addr* parameter indicates which timer expired.

PRU_SLOWTIMO

A "slow timeout" has occured. This request is made when a timeout occurs in the protocol's *pr_slowtimo* routine. The *addr* parameter indicates which timer expired.

PRU_PROTORCV

This request is used in the protocol-protocol interface, not by the routines. It requests reception of data destined for the protocol and not the user. No protocols currently use this facility.

PRU_PROTOSEND

This request allows a protocol to send data destined for another protocol module, not a user. The details of how data is marked "addressed to protocol" instead of "addressed to user" are left to the protocol modules. No protocols currently use this facility.

8. Protocol/protocol interface

The interface between protocol modules is through the *pr_usrreq*, *pr_input*, *pr_output*, *pr_ctlinput*, and *pr_ctloutput* routines. The calling conventions for all but the *pr_usrreq* routine are expected to be specific to the protocol modules and are not guaranteed to be consistent across protocol families. We will examine the conventions used for some of the Internet protocols in this section as an example.

8.1. pr_output

The Internet protocol UDP uses the convention,

    error = udp_output(inp, m);
    int error; struct inpcb *inp; struct mbuf *m;

where the *inp*, "internet protocol control block", passed between modules conveys per connection state information, and the mbuf chain contains the data to be sent. UDP performs consistency checks, appends its header, calculates a checksum, etc. before passing the packet on to the IP module:

    error = ip_output(m, opt, ro, allowbroadcast);
    int error; struct mbuf *m, *opt; struct route *ro; int allowbroadcast;

The call to IP's output routine is more complicated than that for UDP, as befits the additional work the IP module must do. The *m* parameter is the data to be sent, and the *opt* parameter is an optional list of IP options which should be placed in the IP packet header. The *ro* parameter is is used in making routing decisions (and passing them back to the caller). The final parameter, *allowbroadcast* is a flag indicating if the user is allowed to transmit a broadcast packet. This may be inconsequential if the underlying hardware does not support the notion of broadcasting.

All output routines return 0 on success and a UNIX error number if a failure occured which could be immediately detected (no buffer space available, no route to destination, etc.).

8.2. pr_input

Both UDP and TCP use the following calling convention,

    (void) (*protosw[].pr_input)(m);
    struct mbuf *m;

Each mbuf list passed is a single packet to be processed by the protocol module.

The IP input routine is a VAX software interrupt level routine, and so is not called with any parameters. It instead communicates with network interfaces through a queue, *ipintrq*, which is identical in structure to the queues used by the network interfaces for storing packets awaiting transmission.

8.3. pr_ctlinput

This routine is used to convey "control" information to a protocol module (i.e. information which might be passed to the user, but is not data). This routine, and the *pr_ctloutput* routine, have not been extensively developed, and thus suffer from a "clumsiness" that can only be improved as more demands are placed on it.

The common calling convention for this routine is,

    (void) (*protosw[].pr_ctlinput)(req, info);
    int req; caddr_t info;

The *req* parameter is one of the following,

```
#define  PRC_IFDOWN            0     /* interface transition */
#define  PRC_ROUTEDEAD         1     /* select new route if possible */
#define  PRC_QUENCH            4     /* some said to slow down */
#define  PRC_HOSTDEAD          6     /* normally from IMP */
#define  PRC_HOSTUNREACH       7     /* ditto */
#define  PRC_UNREACH_NET       8     /* no route to network */
#define  PRC_UNREACH_HOST      9     /* no route to host */
#define  PRC_UNREACH_PROTOCOL 10     /* dst says bad protocol */
#define  PRC_UNREACH_PORT     11     /* bad port # */
#define  PRC_MSGSIZE          12     /* message size forced drop */
#define  PRC_REDIRECT_NET     13     /* net routing redirect */
#define  PRC_REDIRECT_HOST    14     /* host routing redirect */
#define  PRC_TIMXCEED_INTRANS 17     /* packet lifetime expired in transit */
#define  PRC_TIMXCEED_REASS   18     /* lifetime expired on reass q */
#define  PRC_PARAMPROB        19     /* header incorrect */
```

while the *info* parameter is a "catchall" value which is request dependent. Many of the requests have obviously been derived from ICMP (the Internet *Control Message Protocol*), and from error messages defined in the 1822 host/IMP convention [BBN78]. Mapping tables exist to convert control requests to UNIX error codes which are delivered to a user.

8.4. pr_ctloutput

This routine is not currently used by any protocol modules.

9. Protocol/network-interface interface

The lowest layer in the set of protocols which comprise a protocol family must interface itself to one or more network interfaces in order to transmit and receive packets. It is assumed that any routing decisions have been made before handing a packet to a network interface, in fact this is absolutely necessary in order to locate any interface at all (unless, of course, one uses a single "hardwired" interface). There are two cases to be concerned with, transmission of a packet, and receipt of a packet; each will be considered separately.

9.1. Packet transmission

Assuming a protocol has a handle on an interface, *ifp*, a (struct ifnet *), it transmits a fully formatted packet with the following call,

```
error = (*ifp->if_output)(ifp, m, dst)
int error; struct ifnet *ifp; struct mbuf *m; struct sockaddr *dst;
```

The output routine for the network interface transmits the packet *m* to the *dst* address, or returns an error indication (a UNIX error number). In reality transmission may not be immediate, or successful; normally the output routine simply queues the packet on its send queue and primes an interrupt driven routine to actually transmit the packet. For unreliable mediums, such as the Ethernet, "successful" transmission simply means the packet has been placed on the cable without a collision. On the other hand, an 1822 interface guarantees proper delivery or an error indication for each message transmitted. The model employed in the networking system attaches no promises of delivery to the packets handed to a network interface, and thus corresponds more closely to the Ethernet. Errors returned by the output routine are normally trivial in nature (no buffer space, address format not handled, etc.).

9.2. Packet reception

Each protocol family must have one or more "lowest level" protocols. These protocols deal with internetwork addressing and are responsible for the delivery of incoming packets to the proper protocol processing modules. In the PUP model [Boggs78] these protocols are termed Level 1 protocols, in the ISO model, network layer protocols. In our system each such protocol module has an input packet queue assigned to it. Incoming packets received by a network interface are queued up for the protocol module and a VAX software interrupt is posted to initiate processing.

Three macros are available for queueing and dequeueing packets,

IF_ENQUEUE(ifq, m)
> This places the packet *m* at the tail of the queue *ifq*.

IF_DEQUEUE(ifq, m)
> This places a pointer to the packet at the head of queue *ifq* in *m*. A zero value will be returned in *m* if the queue is empty.

IF_PREPEND(ifq, m)
> This places the packet *m* at the head of the queue *ifq*.

Each queue has a maximum length associated with it as a simple form of congestion control. The macro IF_QFULL(ifq) returns 1 if the queue is filled, in which case the macro IF_DROP(ifq) should be used to bump a count of the number of packets dropped and the offending packet dropped. For example, the following code fragment is commonly found in a network interface's input routine,

```
if (IF_QFULL(inq)) {
        IF_DROP(inq);
        m_freem(m);
} else
        IF_ENQUEUE(inq, m);
```

10. Gateways and routing issues

The system has been designed with the expectation that it will be used in an internetwork environment. The "canonical" environment was envisioned to be a collection of local area networks connected at one or more points through hosts with multiple network interfaces (one on each local area network), and possibly a connection to a long haul network (for example, the ARPANET). In such an environment, issues of gatewaying and packet routing become very important. Certain of these issues, such as congestion control, have been handled in a simplistic manner or specifically not addressed. Instead, where possible, the network system attempts to provide simple mechanisms upon which more involved policies may be implemented. As some of these problems become better understood, the solutions developed will be incorporated into the system.

This section will describe the facilities provided for packet routing. The simplistic mechanisms provided for congestion control are described in chapter 12.

10.1. Routing tables

The network system maintains a set of routing tables for selecting a network interface to use in delivering a packet to its destination. These tables are of the form:

```
struct rtentry {
        u_long    rt_hash;            /* hash key for lookups */
        struct    sockaddr rt_dst;    /* destination net or host */
        struct    sockaddr rt_gateway;/* forwarding agent */
        short     rt_flags;           /* see below */
        short     rt_refcnt;          /* no. of references to structure */
        u_long    rt_use;             /* packets sent using route */
        struct    ifnet *rt_ifp;      /* interface to give packet to */
};
```

The routing information is organized in two separate tables, one for routes to a host and one for routes to a network. The distinction between hosts and networks is necessary so that a single mechanism may be used for both broadcast and multi-drop type networks, and also for networks built from point-to-point links (e.g DECnet [DEC80]).

Each table is organized as a hashed set of linked lists. Two 32-bit hash values are calculated by routines defined for each address family; one based on the destination being a host, and one assuming the target is the network portion of the address. Each hash value is used to locate a hash chain to search (by taking the value modulo the hash table size) and the entire 32-bit value is then used as a key in scanning the list of routes. Lookups are applied first to the routing table for hosts, then to the routing table for networks. If both lookups fail, a final lookup is made for a "wildcard" route (by convention, network 0). By doing this, routes to a specific host on a network may be present as well as routes to the network. This also allows a "fall back" network route to be defined to an "smart" gateway which may then perform more intelligent routing.

Each routing table entry contains a destination (who's at the other end of the route), a gateway to send the packet to, and various flags which indicate the route's status and type (host or network). A count of the number of packets sent using the route is kept for use in deciding between multiple routes to the same destination (see below), and a count of "held references" to the dynamically allocated structure is maintained to insure memory reclamation occurs only when the route is not in use. Finally a pointer to the a network interface is kept; packets sent using the route should be handed to this interface.

Routes are typed in two ways: either as host or network, and as "direct" or "indirect". The host/network distinction determines how to compare the *rt_dst* field during lookup. If the route is to a network, only a packet's destination network is compared to the *rt_dst* entry stored in the table. If the route is to a host, the addresses must match bit for bit.

The distinction between "direct" and "indirect" routes indicates whether the destination is directly connected to the source. This is needed when performing local network encapsulation. If a packet is destined for a peer at a host or network which is not directly connected to the source, the internetwork packet header will indicate the address of the eventual destination, while the local network header will indicate the address of the intervening gateway. Should the destination be directly connected, these addresses are likely to be identical, or a mapping between the two exists. The RTF_GATEWAY flag indicates the route is to an "indirect" gateway agent and the local network header should be filled in from the *rt_gateway* field instead of *rt_dst*, or from the internetwork destination address.

It is assumed multiple routes to the same destination will not be present unless they are deemed *equal* in cost (the current routing policy process never installs multiple routes to the same destination). However, should multiple routes to the same destination exist, a request for a route will return the "least used" route based on the total number of packets sent along this route. This can result in a "ping-pong" effect (alternate packets taking alternate routes), unless protocols "hold onto" routes until they no longer find them useful; either because the destination has changed, or because the route is lossy.

Routing redirect control messages are used to dynamically modify existing routing table entries as well as dynamically create new routing table entries. On hosts where exhaustive routing information is too expensive to maintain (e.g. work stations), the combination of wildcard routing entries and routing redirect messages can be used to provide a simple routing management scheme without the use of a higher level policy process. Statistics are kept by the routing table routines on the use of routing redirect messages and their affect on the routing tables. These statistics may be viewed using

Status information other than routing redirect control messages may be used in the future, but at present they are ignored. Likewise, more intelligent "metrics" may be used to describe routes in the future, possibly based on bandwidth and monetary costs.

10.2. Routing table interface

A protocol accesses the routing tables through three routines, one to allocate a route, one to free a route, and one to process a routing redirect control message. The routine *rtalloc* performs route allocation; it is called with a pointer to the following structure,

```
struct route {
        struct      rtentry *ro_rt;
        struct      sockaddr ro_dst;
};
```

The route returned is assumed "held" by the caller until disposed of with an *rtfree* call. Protocols which implement virtual circuits, such as TCP, hold onto routes for the duration of the circuit's lifetime, while connection-less protocols, such as UDP, currently allocate and free routes on each transmission.

The routine *rtredirect* is called to process a routing redirect control message. It is called with a destination address and the new gateway to that destination. If a non-wildcard route exists to the destination, the gateway entry in the route is modified to point at the new gateway supplied. Otherwise, a new routing table entry is inserted reflecting the information supplied. Routes to interfaces and routes to gateways which are not directly accesible from the host are ignored.

10.3. User level routing policies

Routing policies implemented in user processes manipulate the kernel routing tables through two *ioctl* calls. The commands SIOCADDRT and SIOCDELRT add and delete routing entries, respectively; the tables are read through the /dev/kmem device. The decision to place policy decisions in a user process implies routing table updates may lag a bit behind the identification of new routes, or the failure of existing routes, but this period of instability is

normally very small with proper implementation of the routing process. Advisory information, such as ICMP error messages and IMP diagnostic messages, may be read from raw sockets (described in the next section).

One routing policy process has already been implemented. The system standard "routing daemon" uses a variant of the Xerox NS Routing Information Protocol [Xerox82] to maintain up to date routing tables in our local environment. Interaction with other existing routing protocols, such as the Internet GGP (Gateway-Gateway Protocol), may be accomplished using a similar process.

11. Raw sockets

A raw socket is a mechanism which allows users direct access to a lower level protocol. Raw sockets are intended for knowledgeable processes which wish to take advantage of some protocol feature not directly accessible through the normal interface, or for the development of new protocols built atop existing lower level protocols. For example, a new version of TCP might be developed at the user level by utilizing a raw IP socket for delivery of packets. The raw IP socket interface attempts to provide an identical interface to the one a protocol would have if it were resident in the kernel.

The raw socket support is built around a generic raw socket interface, and (possibly) augmented by protocol-specific processing routines. This section will describe the core of the raw socket interface.

11.1. Control blocks

Every raw socket has a protocol control block of the following form,

```
struct rawcb {
        struct    rawcb *rcb_next;        /* doubly linked list */
        struct    rawcb *rcb_prev;
        struct    socket *rcb_socket;     /* back pointer to socket */
        struct    sockaddr rcb_faddr;     /* destination address */
        struct    sockaddr rcb_laddr;     /* socket's address */
        caddr_t   rcb_pcb;                /* protocol specific stuff */
        short     rcb_flags;
};
```

All the control blocks are kept on a doubly linked list for performing lookups during packet dispatch. Associations may be recorded in the control block and used by the output routine in preparing packets for transmission. The addresses are also used to filter packets on input; this will be described in more detail shortly. If any protocol specific information is required, it may be attached to the control block using the *rcb_pcb* field.

A raw socket interface is datagram oriented. That is, each send or receive on the socket requires a destination address. This address may be supplied by the user or stored in the control block and automatically installed in the outgoing packet by the output routine. Since it is not possible to determine whether an address is present or not in the control block, two flags, RAW_LADDR and RAW_FADDR, indicate if a local and foreign address are present. Another flag, RAW_DONTROUTE, indicates if routing should be performed on outgoing packets. If it is, a route is expected to be allocated for each "new" destination address. That is, the first time a packet is transmitted a route is determined, and thereafter each time the destination address stored in *rcb_route* differs from *rcb_faddr*, or *rcb_route.ro_rt* is zero, the old route is discarded and a new one allocated.

11.2. Input processing

Input packets are "assigned" to raw sockets based on a simple pattern matching scheme. Each network interface or protocol gives packets to the raw input routine with the call:

```
raw_input(m, proto, src, dst)
struct mbuf *m; struct sockproto *proto, struct sockaddr *src, *dst;
```

The data packet then has a generic header prepended to it of the form

```
struct raw_header {
        struct    sockproto raw_proto;
        struct    sockaddr raw_dst;
        struct    sockaddr raw_src;
};
```

and it is placed in a packet queue for the "raw input protocol" module. Packets taken from this queue are copied into any raw sockets that match the header according to the following rules,

1)   The protocol family of the socket and header agree.

2)   If the protocol number in the socket is non-zero, then it agrees with that found in the packet header.

3)   If a local address is defined for the socket, the address format of the local address is the same as the destination address's and the two addresses agree bit for bit.

4)   The rules of 3) are applied to the socket's foreign address and the packet's source address.

A basic assumption is that addresses present in the control block and packet header (as constructed by the network interface and any raw input protocol module) are in a canonical form which may be "block compared".

11.3. Output processing

On output the raw *pr_usrreq* routine passes the packet and raw control block to the raw protocol output routine for any processing required before it is delivered to the appropriate network interface. The output routine is normally the only code required to implement a raw socket interface.

12. Buffering and congestion control

One of the major factors in the performance of a protocol is the buffering policy used. Lack of a proper buffering policy can force packets to be dropped, cause falsified windowing information to be emitted by protocols, fragment host memory, degrade the overall host performance, etc. Due to problems such as these, most systems allocate a fixed pool of memory to the networking system and impose a policy optimized for "normal" network operation.

The networking system developed for UNIX is little different in this respect. At boot time a fixed amount of memory is allocated by the networking system. At later times more system memory may be requested as the need arises, but at no time is memory ever returned to the system. It is possible to garbage collect memory from the network, but difficult. In order to perform this garbage collection properly, some portion of the network will have to be "turned off" as data structures are updated. The interval over which this occurs must kept small compared to the average inter-packet arrival time, or too much traffic may be lost, impacting other hosts on the network, as well as increasing load on the interconnecting mediums. In our environment we have not experienced a need for such compaction, and thus have left the problem unresolved.

The mbuf structure was introduced in chapter 5. In this section a brief description will be given of the allocation mechanisms, and policies used by the protocols in performing connection level buffering.

12.1. Memory management

The basic memory allocation routines place no restrictions on the amount of space which may be allocated. Any request made is filled until the system memory allocator starts refusing to allocate additional memory. When the current quota of memory is insufficient to satisfy an mbuf allocation request, the allocator requests enough new pages from the system to satisfy the current request only. All memory owned by the network is described by a private page table used in remapping pages to be logically contiguous as the need arises. In addition, an array of reference counts parallels the page table and is used when multiple copies of a page are present.

Mbufs are 128 byte structures, 8 fitting in a 1Kbyte page of memory. When data is placed in mbufs, if possible, it is copied or remapped into logically contiguous pages of memory from the network page pool. Data smaller than the size of a page is copied into one or more 112 byte mbuf data areas.

12.2. Protocol buffering policies

Protocols reserve fixed amounts of buffering for send and receive queues at socket creation time. These amounts define the high and low water marks used by the socket routines in deciding when to block and unblock a process. The reservation of space does not currently result in any action by the memory management routines, though it is clear if one imposed an upper bound on the total amount of physical memory allocated to the network, reserving memory would become important.

Protocols which provide connection level flow control do this based on the amount of space in the associated socket queues. That is, send windows are calculated based on the amount of free space in the socket's receive queue, while receive windows are adjusted based on the amount of data awaiting transmission in the send queue. Care has been taken to avoid the "silly window syndrome" described in [Clark82] at both the sending and receiving ends.

12.3. Queue limiting

Incoming packets from the network are always received unless memory allocation fails. However, each Level 1 protocol input queue has an upper bound on the queue's length, and any packets exceeding that bound are discarded. It is possible for a host to be overwhelmed by excessive network traffic (for instance a host acting as a gateway from a high bandwidth network to a low bandwidth network). As a "defensive" mechanism the queue limits may be

adjusted to throttle network traffic load on a host. Consider a host willing to devote some percentage of its machine to handling network traffic. If the cost of handling an incoming packet can be calculated so that an acceptable "packet handling rate" can be determined, then input queue lengths may be dynamically adjusted based on a host's network load and the number of packets awaiting processing. Obviously, discarding packets is not a satisfactory solution to a problem such as this (simply dropping packets is likely to increase the load on a network); the queue lengths were incorporated mainly as a safeguard mechanism.

12.4. Packet forwarding

When packets can not be forwarded because of memory limitations, the system generates a "source quench" message. In addition, any other problems encountered during packet forwarding are also reflected back to the sender in the form of ICMP packets. This helps hosts avoid unneeded retransmissions.

Broadcast packets are never forwarded due to possible dire consequences. In an early stage of network development, broadcast packets were forwarded and a "routing loop" resulted in network saturation and every host on the network crashing.

13. Out of band data

Out of band data is a facility peculiar to the stream socket abstraction defined. Little agreement appears to exist as to what its semantics should be. TCP defines the notion of "urgent data" as in-line, while the NBS protocols [Burruss81] and numerous others provide a fully independent logical transmission channel along which out of band data is to be sent. In addition, the amount of the data which may be sent as an out of band message varies from protocol to protocol; everything from 1 bit to 16 bytes or more.

A stream socket's notion of out of band data has been defined as the lowest reasonable common denominator (at least reasonable in our minds); clearly this is subject to debate. Out of band data is expected to be transmitted out of the normal sequencing and flow control constraints of the data stream. A minimum of 1 byte of out of band data and one outstanding out of band message are expected to be supported by the protocol supporting a stream socket. It is a protocols perogative to support larger sized messages, or more than one outstanding out of band message at a time.

Out of band data is maintained by the protocol and usually not stored in the socket's send queue. The PRU_SENDOOB and PRU_RCVOOB requests to the *pr_usrreq* routine are used in sending and receiving data.

14. Trailer protocols

Core to core copies can be expensive. Consequently, a great deal of effort was spent in minimizing such operations. The VAX architecture provides virtual memory hardware organized in page units. To cut down on copy operations, data is kept in page sized units on page-aligned boundaries whenever possible. This allows data to be moved in memory simply by remapping the page instead of copying. The mbuf and network interface routines perform page table manipulations where needed, hiding the complexities of the VAX virtual memory hardware from higher level code.

Data enters the system in two ways: from the user, or from the network (hardware interface). When data is copied from the user's address space into the system it is deposited in pages (if sufficient data is present to fill an entire page). This encourages the user to transmit information in messages which are a multiple of the system page size.

Unfortunately, performing a similar operation when taking data from the network is very difficult. Consider the format of an incoming packet. A packet usually contains a local network header followed by one or more headers used by the high level protocols. Finally, the data, if any, follows these headers. Since the header information may be variable length, DMA'ing the eventual data for the user into a page aligned area of memory is impossible without a priori knowledge of the format (e.g. supporting only a single protocol header format).

To allow variable length header information to be present and still ensure page alignment of data, a special local network encapsulation may be used. This encapsulation, termed a *trailer protocol*, places the variable length header information after the data. A fixed size local network header is then prepended to the resultant packet. The local network header contains the size of the data portion, and a new *trailer protocol header*, inserted before the variable length information, contains the size of the variable length header information. The following trailer protocol header is used to store information regarding the variable length protocol header:

```
struct {
        short   protocol;       /* original protocol no. */
        short   length;         /* length of trailer */
};
```

The processing of the trailer protocol is very simple. On output, the local network header indicates a trailer encapsulation is being used. The protocol identifier also includes an indication of the number of data pages present (before the trailer protocol header). The trailer protocol header is initialized to contain the actual protocol and variable length header size, and appended to the data along with the variable length header information.

On input, the interface routines identify the trailer encapsulation by the protocol type stored in the local network header, then calculate the number of pages of data to find the beginning of the trailer. The trailing information is copied into a separate mbuf and linked to the front of the resultant packet.

Clearly, trailer protocols require cooperation between source and destination. In addition, they are normally cost effective only when sizable packets are used. The current scheme works because the local network encapsulation header is a fixed size, allowing DMA operations to be performed at a known offset from the first data page being received. Should the local network header be variable length this scheme fails.

Statistics collected indicate as much as 200Kb/s can be gained by using a trailer protocol with 1Kbyte packets. The average size of the variable length header was 40 bytes (the size of a minimal TCP/IP packet header). If hardware supports larger sized packets, even greater gains may be realized.

Acknowledgements

References

[Boggs79]   Boggs, D. R., J. F. Shoch, E. A. Taft, and R. M. Metcalfe; *PUP: An Internetwork Architecture.* Report CSL-79-10. XEROX Palo Alto Research Center, July 1979.

[BBN78]   Bolt Beranek and Newman; *Specification for the Interconnection of Host and IMP.* BBN Technical Report 1822. May 1978.

[Cerf78]   Cerf, V. G.; The Catenet Model for Internetworking. Internet Working Group, IEN 48. July 1978.

[Clark82]   Clark, D. D.; Window and Acknowledgement Strategy in TCP. Internet Working Group, IEN Draft Clark-2. March 1982.

[DEC80]   Digital Equipment Corporation; *DECnet DIGITAL Network Architecture — General Description.* Order No. AA-K179A-TK. October 1980.

[Gurwitz81]   Gurwitz, R. F.; VAX-UNIX Networking Support Project — Implementation Description. Internetwork Working Group, IEN 168. January 1981.

[ISO81]   International Organization for Standardization. *ISO Open Systems Interconnection — Basic Reference Model.* ISO/TC 97/SC 16 N 719. August 1981.

[Joy82a]   Joy, W.; Cooper, E.; Fabry, R.; Leffler, S.; and McKusick, M.; *4.2BSD System Manual.* Computer Systems Research Group, Technical Report 5. University of California, Berkeley. Draft of September 1, 1982.

[Postel79]   Postel, J., ed. *DOD Standard User Datagram Protocol.* Internet Working Group, IEN 88. May 1979.

[Postel80a]   Postel, J., ed. *DOD Standard Internet Protocol.* Internet Working Group, IEN 128. January 1980.

[Postel80b]   Postel, J., ed. *DOD Standard Transmission Control Protocol.* Internet Working Group, IEN 129. January 1980.

[Xerox81]   Xerox Corporation. *Internet Transport Protocols.* Xerox System Integration Standard 028112. December 1981.

[Zimmermann80]   Zimmermann, H. OSI Reference Model — The ISO Model of Architecture for Open Systems Interconnection. IEEE Transactions on Communications. Com-28(4); 425-432. April 1980.

:

SENDMAIL — An Internetwork Mail Router

Eric Allman†

Britton-Lee, Inc.
1919 Addison Street, Suite 105.
Berkeley, California 94704.

ABSTRACT

Routing mail through a heterogenous internet presents many new problems. Among the worst of these is that of address mapping. Historically, this has been handled on an *ad hoc* basis. However, this approach has become unmanageable as internets grow.

Sendmail acts a unified "post office" to which all mail can be submitted. Address interpretation is controlled by a production system, which can parse both domain-based addressing and old-style *ad hoc* addresses. The production system is powerful enough to rewrite addresses in the message header to conform to the standards of a number of common target networks, including old (NCP/RFC733) Arpanet, new (TCP/RFC822) Arpanet, UUCP, and Phonenet. Sendmail also implements an SMTP server, message queueing, and aliasing.

Sendmail implements a general internetwork mail routing facility, featuring aliasing and forwarding, automatic routing to network gateways, and flexible configuration.

In a simple network, each node has an address, and resources can be identified with a host-resource pair; in particular, the mail system can refer to users using a host-username pair. Host names and numbers have to be administered by a central authority, but usernames can be assigned locally to each host.

In an internet, multiple networks with different characteristics and managements must communicate. In particular, the syntax and semantics of resource identification change. Certain special cases can be handled trivially by *ad hoc* techniques, such as providing network names that appear local to hosts on other networks, as with the Ethernet at Xerox PARC. However, the general case is extremely complex. For example, some networks require point-to-point routing, which simplifies the database update problem since only adjacent hosts must be entered into the system tables, while others use end-to-end addressing. Some networks use a left-associative syntax and others use a right-associative syntax, causing ambiguity in mixed addresses.

Internet standards seek to eliminate these problems. Initially, these proposed expanding the address pairs to address triples, consisting of {network, host, resource} triples. Network numbers must be universally agreed upon, and hosts can be assigned locally on each network. The user-level presentation was quickly expanded to address domains, comprised of a local resource identification and a hierarchical domain specification with a common static root. The domain technique separates the issue of physical versus logical addressing. For example, an address of the form "eric@a.cc.berkeley.arpa" describes only the logical organization of the address space.

Sendmail is intended to help bridge the gap between the totally *ad hoc* world of networks that know nothing of each other and the clean, tightly-coupled world of unique network numbers. It can accept old arbitrary address syntaxes, resolving ambiguities using heuristics specified by the system administrator, as well as domain-based addressing. It helps guide the conversion of message formats between disparate networks. In short, *sendmail* is designed to assist a graceful transition to consistent internetwork addressing schemes.

Section 1 discusses the design goals for *sendmail*. Section 2 gives an overview of the basic functions of the system. In section 3, details of usage are discussed. Section 4 compares *sendmail* to other internet mail routers, and an evaluation of *sendmail* is given in section 5, including future plans.

1. DESIGN GOALS

Design goals for *sendmail* include:

(1) Compatibility with the existing mail programs, including Bell version 6 mail, Bell version 7 mail [UNIX83], Berkeley *Mail* [Shoens79], BerkNet mail [Schmidt79], and hopefully UUCP mail [Nowitz78a, Nowitz78b]. ARPANET mail [Crocker77a, Postel77] was also required.

(2) Reliability, in the sense of guaranteeing that every message is correctly delivered or at least brought to the attention of a human for correct disposal; no message should ever be completely lost. This goal was considered essential because of the emphasis on mail in our environment. It has turned out to be one of the hardest goals to satisfy, especially in the face of the many anomalous message formats produced by various ARPANET sites. For example, certain sites generate improperly formated addresses, occasionally causing error-message loops. Some hosts use blanks in names, causing problems with UNIX mail programs that assume that an address is one word. The semantics of some fields are interpreted slightly differently by different sites. In summary, the obscure features of the ARPANET mail protocol really *are* used and are difficult to support, but must be supported.

(3) Existing software to do actual delivery should be used whenever possible. This goal derives as much from political and practical considerations as technical.

(4) Easy expansion to fairly complex environments, including multiple connections to a single network type (such as with multiple UUCP or Ether nets [Metcalfe76]). This goal requires consideration of the contents of an address as well as its syntax in order to determine which gateway to use. For example, the ARPANET is bringing up the TCP protocol to replace the old NCP protocol. No host at Berkeley runs both TCP and NCP, so it is necessary to look at the ARPANET host name to determine whether to route mail to an NCP gateway or a TCP gateway.

(5) Configuration should not be compiled into the code. A single compiled program should be able to run as is at any site (barring such basic changes as the CPU type or the operating system). We have found this seemingly unimportant goal to be critical in real life. Besides the simple problems that occur when any program gets recompiled in a different environment, many sites like to "fiddle" with anything that they will be recompiling anyway.

(6) *Sendmail* must be able to let various groups maintain their own mailing lists, and let individuals specify their own forwarding, without modifying the system alias file.

(7) Each user should be able to specify which mailer to execute to process mail being delivered for him. This feature allows users who are using specialized mailers that use a different format to build their environment without changing the system, and facilitates specialized functions (such as returning an "I am on vacation" message).

(8) Network traffic should be minimized by batching addresses to a single host where possible, without assistance from the user.

These goals motivated the architecture illustrated in figure 1. The user interacts with a mail generating and sending program. When the mail is created, the generator calls *sendmail*, which routes the message to the correct mailer(s). Since some of the senders may be network servers and some of the mailers may be network clients, *sendmail* may be used as an internet mail gateway.

2. OVERVIEW

2.1. System Organization

Sendmail neither interfaces with the user nor does actual mail delivery. Rather, it collects a message generated by a user interface program (UIP) such as Berkeley *Mail*, MS [Crocker77b], or MH [Borden79], edits the message as required by the destination network, and calls appropriate mailers to do mail delivery or queueing for network transmission[1]. This discipline allows the insertion of new mailers at minimum cost. In this sense *sendmail* resembles the Message Processing Module (MPM) of [Postel79b].

2.2. Interfaces to the Outside World

There are three ways *sendmail* can communicate with the outside world, both in receiving and in sending mail. These are using the conventional UNIX argument vector/return status, speaking SMTP over a pair of UNIX pipes, and speaking SMTP over an interprocess(or) channel.

2.2.1. Argument vector/exit status

This technique is the standard UNIX method for communicating with the process. A list of recipients is sent in the argument vector, and the message body is sent
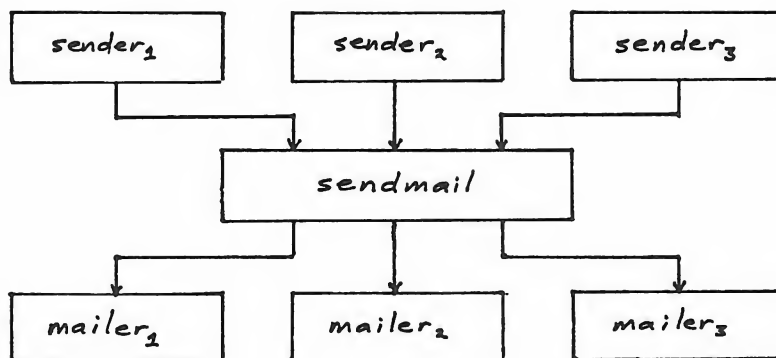

Figure 1 — Sendmail System Structure.

---

[1]except when mailing to a file, when *sendmail* does the delivery directly.

on the standard input. Anything that the mailer prints is simply collected and sent
back to the sender if there were any problems. The exit status from the mailer is col-
lected after the message is sent, and a diagnostic is printed if appropriate.

2.2.2. SMTP over pipes

The SMTP protocol [Postel82] can be used to run an interactive lock-step inter-
face with the mailer. A subprocess is still created, but no recipient addresses are
passed to the mailer via the argument list. Instead, they are passed one at a time in
commands sent to the processes standard input. Anything appearing on the standard
output must be a reply code in a special format.

2.2.3. SMTP over an IPC connection

This technique is similar to the previous technique, except that it uses a 4.2BSD
IPC channel [UNIX83]. This method is exceptionally flexible in that the mailer need
not reside on the same machine. It is normally used to connect to a sendmail process
on another machine.

2.3. Operational Description

When a sender wants to send a message, it issues a request to *sendmail* using one
of the three methods described above. *Sendmail* operates in two distinct phases. In the
first phase, it collects and stores the message. In the second phase, message delivery
occurs. If there were errors during processing during the second phase, *sendmail* creates
and returns a new message describing the error and/or returns an status code telling
what went wrong.

2.3.1. Argument processing and address parsing

If *sendmail* is called using one of the two subprocess techniques, the arguments
are first scanned and option specifications are processed. Recipient addresses are then
collected, either from the command line or from the SMTP RCPT command, and a
list of recipients is created. Aliases are expanded at this step, including mailing lists.
As much validation as possible of the addresses is done at this step: syntax is
checked, and local addresses are verified, but detailed checking of host names and
addresses is deferred until delivery. Forwarding is also performed as the local
addresses are verified.

Sendmail appends each address to the recipient list after parsing. When a name
is aliased or forwarded, the old name is retained in the list, and a flag is set that tells
the delivery phase to ignore this recipient. This list is kept free from duplicates,
preventing alias loops and duplicate messages deliverd to the same recipient, as might
occur if a person is in two groups.

2.3.2. Message collection

Sendmail then collects the message. The message should have a header at the
beginning. No formatting requirements are imposed on the message except that they
must be lines of text (i.e., binary data is not allowed). The header is parsed and
stored in memory, and the body of the message is saved in a temporary file.

To simplify the program interface, the message is collected even if no addresses
were valid. The message will be returned with an error.

2.3.3. Message delivery

For each unique mailer and host in the recipient list, *sendmail* calls the appropri-
ate mailer. Each mailer invocation sends to all users receiving the message on one
host. Mailers that only accept one recipient at a time are handled properly.

The message is sent to the mailer using one of the same three interfaces used to submit a message to sendmail. Each copy of the message is prepended by a customized header. The mailer status code is caught and checked, and a suitable error message given as appropriate. The exit code must conform to a system standard or a generic message ("Service unavailable") is given.

2.3.4. Queueing for retransmission

If the mailer returned an status that indicated that it might be able to handle the mail later, *sendmail* will queue the mail and try again later.

2.3.5. Return to sender

If errors occur during processing, *sendmail* returns the message to the sender for retransmission. The letter can be mailed back or written in the file "dead.letter" in the sender's home directory[2].

2.4. Message Header Editing

Certain editing of the message header occurs automatically. Header lines can be inserted under control of the configuration file. Some lines can be merged; for example, a "From:" line and a "Full-name:" line can be merged under certain circumstances.

2.5. Configuration File

Almost all configuration information is read at runtime from an ASCII file, encoding macro definitions (defining the value of macros used internally), header declarations (telling sendmail the format of header lines that it will process specially, i.e., lines that it will add or reformat), mailer definitions (giving information such as the location and characteristics of each mailer), and address rewriting rules (a limited production system to rewrite addresses which is used to parse and rewrite the addresses).

To improve performance when reading the configuration file, a memory image can be provided. This provides a "compiled" form of the configuration file.

3. USAGE AND IMPLEMENTATION

3.1. Arguments

Arguments may be flags and addresses. Flags set various processing options. Following flag arguments, address arguments may be given, unless we are running in SMTP mode. Addresses follow the syntax in RFC822 [Crocker82] for ARPANET address formats. In brief, the format is:

(1) Anything in parentheses is thrown away (as a comment).

(2) Anything in angle brackets ("< >") is preferred over anything else. This rule implements the ARPANET standard that addresses of the form

          user name <machine-address>

     will send to the electronic "machine-address" rather than the human "user name."

(3) Double quotes (") quote phrases; backslashes quote characters. Backslashes are more powerful in that they will cause otherwise equivalent phrases to compare differently — for example, *user* and "*user*" are equivalent, but *user* is different from either of them.

---

[2]Obviously, if the site giving the error is not the originating site, the only reasonable option is to mail back to the sender. Also, there are many more error disposition options, but they only effect the error message — the "return to sender" function is always handled in one of these two ways.

Parentheses, angle brackets, and double quotes must be properly balanced and nested. The rewriting rules control remaining parsing[3].

3.2. Mail to Files and Programs

Files and programs are legitimate message recipients. Files provide archival storage of messages, useful for project administration and history. Programs are useful as recipients in a variety of situations, for example, to maintain a public repository of systems messages (such as the Berkeley *msgs* program, or the MARS system [Sattley78]).

Any address passing through the initial parsing algorithm as a local address (i.e, not appearing to be a valid address for another mailer) is scanned for two special cases. If prefixed by a vertical bar ("|") the rest of the address is processed as a shell command. If the user name begins with a slash mark ("/") the name is used as a file name, instead of a login name.

Files that have setuid or setgid bits set but no execute bits set have those bits honored if *sendmail* is running as root.

3.3. Aliasing, Forwarding, Inclusion

Sendmail reroutes mail three ways. Aliasing applies system wide. Forwarding allows each user to reroute incoming mail destined for that account. Inclusion directs *sendmail* to read a file for a list of addresses, and is normally used in conjunction with aliasing.

3.3.1. Aliasing

Aliasing maps names to address lists using a system-wide file. This file is indexed to speed access. Only names that parse as local are allowed as aliases; this guarantees a unique key (since there are no nicknames for the local host).

3.3.2. Forwarding

After aliasing, recipients that are local and valid are checked for the existence of a ".forward" file in their home directory. If it exists, the message is *not* sent to that user, but rather to the list of users in that file. Often this list will contain only one address, and the feature will be used for network mail forwarding.

Forwarding also permits a user to specify a private incoming mailer. For example, forwarding to:

"|/usr/local/newmail myname"

will use a different incoming mailer.

3.3.3. Inclusion

Inclusion is specified in RFC 733 [Crocker77a] syntax:

:Include: pathname

An address of this form reads the file specified by *pathname* and sends to all users listed in that file.

The intent is *not* to support direct use of this feature, but rather to use this as a subset of aliasing. For example, an alias of the form:

project: :include:/usr/project/userlist

is a method of letting a project maintain a mailing list without interaction with the system administration, even if the alias file is protected.

---

[3]Disclaimer: Some special processing is done after rewriting local names; see below.

It is not necessary to rebuild the index on the alias database when a :include: list is changed.

3.4. Message Collection

Once all recipient addresses are parsed and verified, the message is collected. The message comes in two parts: a message header and a message body, separated by a blank line.

The header is formatted as a series of lines of the form

    field-name: field-value

Field-value can be split across lines by starting the following lines with a space or a tab. Some header fields have special internal meaning, and have appropriate special processing. Other headers are simply passed through. Some header fields may be added automatically, such as time stamps.

The body is a series of text lines. It is completely uninterpreted and untouched, except that lines beginning with a dot have the dot doubled when transmitted over an SMTP channel. This extra dot is stripped by the receiver.

3.5. Message Delivery

The send queue is ordered by receiving host before transmission to implement message batching. Each address is marked as it is sent so rescanning the list is safe. An argument list is built as the scan proceeds. Mail to files is detected during the scan of the send list. The interface to the mailer is performed using one of the techniques described in section 2.2.

After a connection is established, *sendmail* makes the per-mailer changes to the header and sends the result to the mailer. If any mail is rejected by the mailer, a flag is set to invoke the return-to-sender function after all delivery completes.

3.6. Queued Messages

If the mailer returns a "temporary failure" exit status, the message is queued. A control file is used to describe the recipients to be sent to and various other parameters. This control file is formatted as a series of lines, each describing a sender, a recipient, the time of submission, or some other salient parameter of the message. The header of the message is stored in the control file, so that the associated data file in the queue is just the temporary file that was originally collected.

3.7. Configuration

Configuration is controlled primarily by a configuration file read at startup. *Sendmail* should not need to be recompiled except

(1) To change operating systems (V6, V7/32V, 4BSD).

(2) To remove or insert the DBM (UNIX database) library.

(3) To change ARPANET reply codes.

(4) To add headers fields requiring special processing.

Adding mailers or changing parsing (i.e., rewriting) or routing information does not require recompilation.

If the mail is being sent by a local user, and the file ".mailcf" exists in the sender's home directory, that file is read as a configuration file after the system configuration file. The primary use of this feature is to add header lines.

The configuration file encodes macro definitions, header definitions, mailer definitions, rewriting rules, and options.

3.7.1. Macros

Macros can be used in three ways. Certain macros transmit unstructured textual information into the mail system, such as the name *sendmail* will use to identify itself in error messages. Other macros transmit information from *sendmail* to the configuration file for use in creating other fields (such as argument vectors to mailers); e.g., the name of the sender, and the host and user of the recipient. Other macros are unused internally, and can be used as shorthand in the configuration file.

3.7.2. Header declarations

Header declarations inform *sendmail* of the format of known header lines. Knowledge of a few header lines is built into *sendmail*, such as the "From:" and "Date:" lines.

Most configured headers will be automatically inserted in the outgoing message if they don't exist in the incoming message. Certain headers are suppressed by some mailers.

3.7.3. Mailer declarations

Mailer declarations tell *sendmail* of the various mailers available to it. The definition specifies the internal name of the mailer, the pathname of the program to call, some flags associated with the mailer, and an argument vector to be used on the call; this vector is macro-expanded before use.

3.7.4. Address rewriting rules

The heart of address parsing in *sendmail* is a set of rewriting rules. These are an ordered list of pattern-replacement rules, (somewhat like a production system, except that order is critical), which are applied to each address. The address is rewritten textually until it is either rewritten into a special canonical form (i.e., a (mailer, host, user) 3-tuple, such as {arpanet, usc-isif, postel} representing the address "postel@usc-isif"), or it falls off the end. When a pattern matches, the rule is reapplied until it fails.

The configuration file also supports the editing of addresses into different formats. For example, an address of the form:

    ucsfcgl!tef

might be mapped into:

    tef@ucsfcgl.UUCP

to conform to the domain syntax. Translations can also be done in the other direction.

3.7.5. Option setting

There are several options that can be set from the configuration file. These include the pathnames of various support files, timeouts, default modes, etc.

4. COMPARISON WITH OTHER MAILERS

4.1. Delivermail

Sendmail is an outgrowth of *delivermail*. The primary differences are:

(1) Configuration information is not compiled in. This change simplifies many of the problems of moving to other machines. It also allows easy debugging of new mailers.

(2) Address parsing is more flexible. For example, *delivermail* only supported one gateway to any network, whereas *sendmail* can be sensitive to host names and reroute to different gateways.

(3) Forwarding and :include: features eliminate the requirement that the system alias file be writable by any user (or that an update program be written, or that the system administration make all changes).

(4) *Sendmail* supports message batching across networks when a message is being sent to multiple recipients.

(5) A mail queue is provided in *sendmail.* Mail that cannot be delivered immediately but can potentially be delivered later is stored in this queue for a later retry. The queue also provides a buffer against system crashes; after the message has been collected it may be reliably redelivered even if the system crashes during the initial delivery.

(6) *Sendmail* uses the networking support provided by 4.2BSD to provide a direct interface networks such as the ARPANET and/or Ethernet using SMTP (the Simple Mail Transfer Protocol) over a TCP/IP connection.

4.2. MMDF

MMDF [Crocker79] spans a wider problem set than *sendmail.* For example, the domain of MMDF includes a "phone network" mailer, whereas *sendmail* calls on preexisting mailers in most cases.

MMDF and *sendmail* both support aliasing, customized mailers, message batching, automatic forwarding to gateways, queueing, and retransmission. MMDF supports two-stage timeout, which *sendmail* does not support.

The configuration for MMDF is compiled into the code[4].

Since MMDF does not consider backwards compatibility as a design goal, the address parsing is simpler but much less flexible.

It is somewhat harder to integrate a new channel[5] into MMDF. In particular, MMDF must know the location and format of host tables for all channels, and the channel must speak a special protocol. This allows MMDF to do additional verification (such as verifying host names) at submission time.

MMDF strictly separates the submission and delivery phases. Although *sendmail* has the concept of each of these stages, they are integrated into one program, whereas in MMDF they are split into two programs.

4.3. Message Processing Module

The Message Processing Module (MPM) discussed by Postel [Postel79b] matches *sendmail* closely in terms of its basic architecture. However, like MMDF, the MPM includes the network interface software as part of its domain.

MPM also postulates a duplex channel to the receiver, as does MMDF, thus allowing simpler handling of errors by the mailer than is possible in *sendmail.* When a message queued by *sendmail* is sent, any errors must be returned to the sender by the mailer itself. Both MPM and MMDF mailers can return an immediate error response, and a single error processor can create an appropriate response.

MPM prefers passing the message as a structured object, with type-length-value

---

[4]Dynamic configuration tables are currently being considered for MMDF; allowing the installer to select either compiled or dynamic tables.

[5]The MMDF equivalent of a *sendmail* "mailer."

tuples[6]. Such a convention requires a much higher degree of cooperation between mailers than is required by *sendmail*. MPM also assumes a universally agreed upon internet name space (with each address in the form of a net-host-user tuple), which *sendmail* does not.

5. EVALUATIONS AND FUTURE PLANS

Sendmail is designed to work in a nonhomogeneous environment. Every attempt is made to avoid imposing unnecessary constraints on the underlying mailers. This goal has driven much of the design. One of the major problems has been the lack of a uniform address space, as postulated in [Postel79a] and [Postel79b].

A nonuniform address space implies that a path will be specified in all addresses, either explicitly (as part of the address) or implicitly (as with implied forwarding to gateways). This restriction has the unpleasant effect of making replying to messages exceedingly difficult, since there is no one "address" for any person, but only a way to get there from wherever you are.

Interfacing to mail programs that were not initially intended to be applied in an internet environment has been amazingly successful, and has reduced the job to a manageable task.

Sendmail has knowledge of a few difficult environments built in. It generates ARPANET FTP/SMTP compatible error messages (prepended with three-digit numbers [Neigus73, Postel74, Postel82]) as necessary, optionally generates UNIX-style "From" lines on the front of messages for some mailers, and knows how to parse the same lines on input. Also, error handling has an option customized for BerkNet.

The decision to avoid doing any type of delivery where possible (even, or perhaps especially, local delivery) has turned out to be a good idea. Even with local delivery, there are issues of the location of the mailbox, the format of the mailbox, the locking protocol used, etc., that are best decided by other programs. One surprisingly major annoyance in many internet mailers is that the location and format of local mail is built in. The feeling seems to be that local mail is so common that it should be efficient. This feeling is not born out by our experience; on the contrary, the location and format of mailboxes seems to vary widely from system to system.

The ability to automatically generate a response to incoming mail (by forwarding mail to a program) seems useful ("I am on vacation until late August....") but can create problems such as forwarding loops (two people on vacation whose programs send notes back and forth, for instance) if these programs are not well written. A program could be written to do standard tasks correctly, but this would solve the general case.

It might be desirable to implement some form of load limiting. I am unaware of any mail system that addresses this problem, nor am I aware of any reasonable solution at this time.

The configuration file is currently practically inscrutable; considerable convenience could be realized with a higher-level format.

It seems clear that common protocols will be changing soon to accommodate changing requirements and environments. These changes will include modifications to the message header (e.g., [NBS80]) or to the body of the message itself (such as for multimedia messages [Postel80]). Experience indicates that these changes should be relatively trivial to integrate into the existing system.

In tightly coupled environments, it would be nice to have a name server such as Grapvine [Birrell82] integrated into the mail system. This would allow a site such as "Berkeley" to appear as a single host, rather than as a collection of hosts, and would allow people to move transparently among machines without having to change their addresses. Such a

---

[6]This is similar to the NBS standard.

facility would require an automatically updated database and some method of resolving conflicts. Ideally this would be effective even without all hosts being under a single management. However, it is not clear whether this feature should be integrated into the aliasing facility or should be considered a "value added" feature outside *sendmail* itself.

As a more interesting case, the CSNET name server [Solomon81] provides an facility that goes beyond a single tightly-coupled environment. Such a facility would normally exist outside of *sendmail* however.

ACKNOWLEDGEMENTS

Thanks are due to Kurt Shoens for his continual cheerful assistance and good advice, Bill Joy for pointing me in the correct direction (over and over), and Mark Horton for more advice, prodding, and many of the good ideas. Kurt and Eric Schmidt are to be credited for using *delivermail* as a server for their programs (*Mail* and BerkNet respectively) before any sane person should have, and making the necessary modifications promptly and happily. Eric gave me considerable advice about the perils of network software which saved me an unknown amount of work and grief. Mark did the original implementation of the DBM version of aliasing, installed the VFORK code, wrote the current version of *rmail*, and was the person who really convinced me to put the work into *delivermail* to turn it into *sendmail*. Kurt deserves accolades for using *sendmail* when I was myself afraid to take the risk; how a person can continue to be so enthusiastic in the face of so much bitter reality is beyond me.

Kurt, Mark, Kirk McKusick, Marvin Solomon, and many others have reviewed this paper, giving considerable useful advice.

Special thanks are reserved for Mike Stonebraker at Berkeley and Bob Epstein at Britton-Lee, who both knowingly allowed me to put so much work into this project when there were so many other things I really should have been working on.

REFERENCES

[Birrell82]   Birrell, A. D., Levin, R., Needham, R. M., and Schroeder, M. D., "Grapevine: An Exercise in Distributed Computing." In *Comm. A.C.M. 25,* 4, April 82.

[Borden79]   Borden, S., Gaines, R. S., and Shapiro, N. Z., *The MH Message Handling System: Users' Manual.* R-2367-PAF. Rand Corporation. October 1979.

[Crocker77a]   Crocker, D. H., Vittal, J. J., Pogran, K. T., and Henderson, D. A. Jr., *Standard for the Format of ARPA Network Text Messages.* RFC 733, NIC 41952. In [Feinler78]. November 1977.

[Crocker77b]   Crocker, D. H., *Framework and Functions of the MS Personal Message System.* R-2134-ARPA, Rand Corporation, Santa Monica, California. 1977.

[Crocker79]   Crocker, D. H., Szurkowski, E. S., and Farber, D. J., *An Internetwork Memo Distribution Facility — MMDF.* 6th Data Communication Symposium, Asilomar. November 1979.

[Crocker82]   Crocker, D. H., *Standard for the Format of Arpa Internet Text Messages.* RFC 822. Network Information Center, SRI International, Menlo Park, California. August 1982.

[Metcalfe76]   Metcalfe, R., and Boggs, D., "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM 19,* 7. July 1976.

[Feinler78]   Feinler, E., and Postel, J. (eds.), *ARPANET Protocol Handbook.* NIC 7104, Network Information Center, SRI International, Menlo Park, California. 1978.

[NBS80]   National Bureau of Standards, *Specification of a Draft Message Format Standard.* Report No. ICST/CBOS 80-2. October 1980.

[Neigus73]   Neigus, N., *File Transfer Protocol for the ARPA Network.* RFC 542, NIC 17759. In [Feinler78]. August, 1973.

[Nowitz78a]   Nowitz, D. A., and Lesk, M. E., *A Dial-Up Network of UNIX Systems.* Bell Laboratories. In UNIX Programmer's Manual, Seventh Edition, Volume 2. August, 1978.

[Nowitz78b]   Nowitz, D. A., *Uucp Implementation Description.* Bell Laboratories. In UNIX Programmer's Manual, Seventh Edition, Volume 2. October, 1978.

[Postel74]   Postel, J., and Neigus, N., Revised FTP Reply Codes. RFC 640, NIC 30843. In [Feinler78]. June, 1974.

[Postel77]   Postel, J., *Mail Protocol.* NIC 29588. In [Feinler78]. November 1977.

[Postel79a]   Postel, J., *Internet Message Protocol.* RFC 753, IEN 85. Network Information Center, SRI International, Menlo Park, California. March 1979.

[Postel79b]   Postel, J. B., *An Internetwork Message Structure.* In *Proceedings of the Sixth Data Communications Symposium,* IEEE. New York. November 1979.

[Postel80]   Postel, J. B., *A Structured Format for Transmission of Multi-Media Documents.* RFC 767. Network Information Center, SRI International, Menlo Park, California. August 1980.

[Postel82]   Postel, J. B., *Simple Mail Transfer Protocol.* RFC821 (obsoleting RFC788). Network Information Center, SRI International, Menlo Park, California. August 1982.

[Schmidt79]   Schmidt, E., *An Introduction to the Berkeley Network.* University of California, Berkeley California. 1979.

[Shoens79]   Shoens, K., *Mail Reference Manual.* University of California, Berkeley. In UNIX Programmer's Manual, Seventh Edition, Volume 2C. December 1979.

[Sluizer81]   Sluizer, S., and Postel, J. B., *Mail Transfer Protocol.* RFC 780. Network Information Center, SRI International, Menlo Park, California. May 1981.

[Solomon81]   Solomon, M., Landweber, L., and Neuhengen, D., "The Design of the CSNET Name Server." CS-DN-2, University of Wisconsin, Madison. November 1981.

[Su82]   Su, Zaw-Sing, and Postel, Jon, *The Domain Naming Convention for Internet User Applications.* RFC819. Network Information Center, SRI International, Menlo Park, California. August 1982.

[UNIX83]   *The UNIX Programmer's Manual, Seventh Edition,* Virtual VAX-11 Version, Volume 1. Bell Laboratories, modified by the University of California, Berkeley, California. March, 1983.

On the Security of UNIX

Dennis M. Ritchie

Recently there has been much interest in the security aspects of operating systems and software. At issue is the ability to prevent undesired disclosure of information, destruction of information, and harm to the functioning of the system. This paper discusses the degree of security which can be provided under the UNIX† system and offers a number of hints on how to improve security.

The first fact to face is that UNIX was not developed with security, in any realistic sense, in mind; this fact alone guarantees a vast number of holes. (Actually the same statement can be made with respect to most systems.) The area of security in which UNIX is theoretically weakest is in protecting against crashing or at least crippling the operation of the system. The problem here is not mainly in uncritical acceptance of bad parameters to system calls— there may be bugs in this area, but none are known—but rather in lack of checks for excessive consumption of resources. Most notably, there is no limit on the amount of disk storage used, either in total space allocated or in the number of files or directories. Here is a particularly ghastly shell sequence guaranteed to stop the system:

```
while : ; do
      mkdir x
      cd x
done
```

Either a panic will occur because all the i-nodes on the device are used up, or all the disk blocks will be consumed, thus preventing anyone from writing files on the device.

In this version of the system, users are prevented from creating more than a set number of processes simultaneously, so unless users are in collusion it is unlikely that any one can stop the system altogether. However, creation of 20 or so CPU or disk-bound jobs leaves few resources available for others. Also, if many large jobs are run simultaneously, swap space may run out, causing a panic.

It should be evident that excessive consumption of disk space, files, swap space, and processes can easily occur accidentally in malfunctioning programs as well as at command level. In fact UNIX is essentially defenseless against this kind of abuse, nor is there any easy fix. The best that can be said is that it is generally fairly easy to detect what has happened when disaster strikes, to identify the user responsible, and take appropriate action. In practice, we have found that difficulties in this area are rather rare, but we have not been faced with malicious users, and enjoy a fairly generous supply of resources which have served to cushion us against accidental overconsumption.

The picture is considerably brighter in the area of protection of information from unauthorized perusal and destruction. Here the degree of security seems

---

† UNIX is a trademark of Bell Laboratories.

(almost) adequate theoretically, and the problems lie more in the necessity for care in the actual use of the system.

Each UNIX file has associated with it eleven bits of protection information together with a user identification number and a user-group identification number (UID and GID). Nine of the protection bits are used to specify independently permission to read, to write, and to execute the file to the user himself, to members of the user's group, and to all other users. Each process generated by or for a user has associated with it an effective UID and a real UID, and an effective and real GID. When an attempt is made to access the file for reading, writing, or execution, the user process's effective UID is compared against the file's UID; if a match is obtained, access is granted provided the read, write, or execute bit respectively for the user himself is present. If the UID for the file and for the process fail to match, but the GID's do match, the group bits are used; if the GID's do not match, the bits for other users are tested. The last two bits of each file's protection information, called the set-UID and set-GID bits, are used only when the file is executed as a program. If, in this case, the set-UID bit is on for the file, the effective UID for the process is changed to the UID associated with the file; the change persists until the process terminates or until the UID changed again by another execution of a set-UID file. Similarly the effective group ID of a process is changed to the GID associated with a file when that file is executed and has the set-GID bit set. The real UID and GID of a process do not change when any file is executed, but only as the result of a privileged system call.

The basic notion of the set-UID and set-GID bits is that one may write a program which is executable by others and which maintains files accessible to others only by that program. The classical example is the game-playing program which maintains records of the scores of its players. The program itself has to read and write the score file, but no one but the game's sponsor can be allowed unrestricted access to the file lest they manipulate the game to their own advantage. The solution is to turn on the set-UID bit of the game program. When, and only when, it is invoked by players of the game, it may update the score file but ordinary programs executed by others cannot access the score.

There are a number of special cases involved in determining access permissions. Since executing a directory as a program is a meaningless operation, the execute-permission bit, for directories, is taken instead to mean permission to search the directory for a given file during the scanning of a path name; thus if a directory has execute permission but no read permission for a given user, he may access files with known names in the directory, but may not read (list) the entire contents of the directory. Write permission on a directory is interpreted to mean that the user may create and delete files in that directory; it is impossible for any user to write directly into any directory.

Another, and from the point of view of security, much more serious special case is that there is a "super user" who is able to read any file and write any non-directory. The super-user is also able to change the protection mode and the owner UID and GID of any file and to invoke privileged system calls. It must be recognized that the mere notion of a super-user is a theoretical, and usually practical, blemish on any protection scheme.

The first necessity for a secure system is of course arranging that all files and directories have the proper protection modes. Traditionally, UNIX software has been exceedingly permissive in this regard; essentially all commands create files readable and writable by everyone. In the current version, this policy may be easily adjusted to suit the needs of the installation or the individual user. Associated with each process and its descendants is a mask, which is in effect

and-ed with the mode of every file and directory created by that process. In this way, users can arrange that, by default, all their files are no more accessible than they wish. The standard mask, set by *login,* allows all permissions to the user himself and to his group, but disallows writing by others.

To maintain both data privacy and data integrity, it is necessary, and largely sufficient, to make one's files inaccessible to others. The lack of sufficiency could follow from the existence of set-UID programs created by the user and the possibility of total breach of system security in one of the ways discussed below (or one of the ways not discussed below). For greater protection, an encryption scheme is available. Since the editor is able to create encrypted documents, and the *crypt* command can be used to pipe such documents into the other text-processing programs, the length of time during which cleartext versions need be available is strictly limited. The encryption scheme used is not one of the strongest known, but it is judged adequate, in the sense that cryptanalysis is likely to require considerably more effort than more direct methods of reading the encrypted files. For example, a user who stores data that he regards as truly secret should be aware that he is implicitly trusting the system administrator not to install a version of the crypt command that stores every typed password in a file.

Needless to say, the system administrators must be at least as careful as their most demanding user to place the correct protection mode on the files under their control. In particular, it is necessary that special files be protected from writing, and probably reading, by ordinary users when they store sensitive files belonging to other users. It is easy to write programs that examine and change files by accessing the device on which the files live.

On the issue of password security, UNIX is probably better than most systems. Passwords are stored in an encrypted form which, in the absence of serious attention from specialists in the field, appears reasonably secure, provided its limitations are understood. In the current version, it is based on a slightly defective version of the Federal DES; it is purposely defective so that easily-available hardware is useless for attempts at exhaustive key-search. Since both the encryption algorithm and the encrypted passwords are available, exhaustive enumeration of potential passwords is still feasible up to a point. We have observed that users choose passwords that are easy to guess: they are short, or from a limited alphabet, or in a dictionary. Passwords should be at least six characters long and randomly chosen from an alphabet which includes digits and special characters.

Of course there also exist feasible non-cryptanalytic ways of finding out passwords. For example: write a program which types out "login:" on the typewriter and copies whatever is typed to a file of your own. Then invoke the command and go away until the victim arrives.

The set-UID (set-GID) notion must be used carefully if any security is to be maintained. The first thing to keep in mind is that a writable set-UID file can have another program copied onto it. For example, if the super-user *(su)* command is writable, anyone can copy the shell onto it and get a password-free version of *su.* A more subtle problem can come from set-UID programs which are not sufficiently careful of what is fed into them. To take an obsolete example, the previous version of the *mail* command was set-UID and owned by the super-user. This version sent mail to the recipient's own directory. The notion was that one should be able to send mail to anyone even if they want to protect their directories from writing. The trouble was that *mail* was rather dumb: anyone could mail someone else's private file to himself. Much more serious is the following scenario: make a file with a line like one in the password file which allows

one to log in as the super-user. Then make a link named ".mail" to the password file in some writable directory on the same device as the password file (say /tmp). Finally mail the bogus login line to /tmp/.mail; You can then login as the super-user, clean up the incriminating evidence, and have your will.

The fact that users can mount their own disks and tapes as file systems can be another way of gaining super-user status. Once a disk pack is mounted, the system believes what is on it. Thus one can take a blank disk pack, put on it anything desired, and mount it. There are obvious and unfortunate consequences. For example: a mounted disk with garbage on it will crash the system; one of the files on the mounted disk can easily be a password-free version of *su;* other files can be unprotected entries for special files. The only easy fix for this problem is to forbid the use of *mount* to unprivileged users. A partial solution, not so restrictive, would be to have the *mount* command examine the special file for bad data, set-UID programs owned by others, and accessible special files, and balk at unprivileged invokers.

Password Security: A Case History

Robert Morris

Ken Thompson

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes the history of the design of the password security scheme on a remotely accessed time-sharing system. The present design was the result of countering observed attempts to penetrate the system. The result is a compromise between extreme security and ease of use.

April 3, 1978

Password Security: A Case History

Robert Morris

Ken Thompson

Bell Laboratories
Murray Hill, New Jersey 07974

INTRODUCTION

Password security on the UNIX† time-sharing system [1] is provided by a collection of programs whose elaborate and strange design is the outgrowth of many years of experience with earlier versions. To help develop a secure system, we have had a continuing competition to devise new ways to attack the security of the system (the bad guy) and, at the same time, to devise new techniques to resist the new attacks (the good guy). This competition has been in the same vein as the competition of long standing between manufacturers of armor plate and those of armor-piercing shells. For this reason, the description that follows will trace the history of the password system rather than simply presenting the program in its current state. In this way, the reasons for the design will be made clearer, as the design cannot be understood without also understanding the potential attacks.

An underlying goal has been to provide password security at minimal inconvenience to the users of the system. For example, those who want to run a completely open system without passwords, or to have passwords only at the option of the individual users, are able to do so, while those who require all of their users to have passwords gain a high degree of security against penetration of the system by unauthorized users.

The password system must be able not only to prevent any access to the system by unauthorized users (i.e. prevent them from logging in at all), but it must also prevent users who are already logged in from doing things that they are not authorized to do. The so called "super-user" password, for example, is especially critical because the super-user has all sorts of permissions and has essentially unlimited access to all system resources.

Password security is of course only one component of overall system security, but it is an essential component. Experience has shown that attempts to penetrate remote-access systems have been astonishingly sophisticated.

Remote-access systems are peculiarly vulnerable to penetration by outsiders as there are threats at the remote terminal, along the communications link, as well as at the computer itself. Although the security of a password encryption algorithm is an interesting intellectual and mathematical problem, it is only one tiny facet of a very large problem. In practice, physical security of the computer, communications security of the communications link, and physical control of the computer itself loom as far more important issues. Perhaps most important of all is control over the actions of ex-employees, since they are not under any direct control and they may have intimate knowledge about the system, its resources, and methods of access. Good system security involves realistic evaluation of the risks not only of deliberate attacks but also of casual unauthorized access and accidental disclosure.

---

†UNIX is a Trademark of Bell Laboratories.

PROLOGUE

The UNIX system was first implemented with a password file that contained the actual passwords of all the users, and for that reason the password file had to be heavily protected against being either read or written. Although historically, this had been the technique used for remote-access systems, it was completely unsatisfactory for several reasons.

The technique is excessively vulnerable to lapses in security. Temporary loss of protection can occur when the password file is being edited or otherwise modified. There is no way to prevent the making of copies by privileged users. Experience with several earlier remote-access systems showed that such lapses occur with frightening frequency. Perhaps the most memorable such occasion occurred in the early 60's when a system administrator on the CTSS system at MIT was editing the password file and another system administrator was editing the daily message that is printed on everyone's terminal on login. Due to a software design error, the temporary editor files of the two users were interchanged and thus, for a time, the password file was printed on every terminal when it was logged in.

Once such a lapse in security has been discovered, everyone's password must be changed, usually simultaneously, at a considerable administrative cost. This is not a great matter, but far more serious is the high probability of such lapses going unnoticed by the system administrators.

Security against unauthorized disclosure of the passwords was, in the last analysis, impossible with this system because, for example, if the contents of the file system are put on to magnetic tape for backup, as they must be, then anyone who has physical access to the tape can read anything on it with no restriction.

Many programs must get information of various kinds about the users of the system, and these programs in general should have no special permission to read the password file. The information which should have been in the password file actually was distributed (or replicated) into a number of files, all of which had to be updated whenever a user was added to or dropped from the system.

THE FIRST SCHEME

The obvious solution is to arrange that the passwords not appear in the system at all, and it is not difficult to decide that this can be done by encrypting each user's password, putting only the encrypted form in the password file, and throwing away his original password (the one that he typed in). When the user later tries to log in to the system, the password that he types is encrypted and compared with the encrypted version in the password file. If the two match, his login attempt is accepted. Such a scheme was first described in [3, p.91ff.]. It also seemed advisable to devise a system in which neither the password file nor the password program itself needed to be protected against being read by anyone.

All that was needed to implement these ideas was to find a means of encryption that was very difficult to invert, even when the encryption program is available. Most of the standard encryption methods used (in the past) for encryption of messages are rather easy to invert. A convenient and rather good encryption program happened to exist on the system at the time: it simulated the M-209 cipher machine [4] used by the U.S. Army during World War II. It turned out that the M-209 program was usable, but with a given key, the ciphers produced by this program are trivial to invert. It is a much more difficult matter to find out the key given the cleartext input and the enciphered output of the program. Therefore, the password was used not as the text to be encrypted but as the key, and a constant was encrypted using this key. The encrypted result was entered into the password file.

ATTACKS ON THE FIRST APPROACH

Suppose that the bad guy has available the text of the password encryption program and the complete password file. Suppose also that he has substantial computing capacity at his disposal.

One obvious approach to penetrating the password mechanism is to attempt to find a general method of inverting the encryption algorithm. Very possibly this can be done, but few successful results have come to light, despite substantial efforts extending over a period of more than five years. The results have not proved to be very useful in penetrating systems.

Another approach to penetration is simply to keep trying potential passwords until one succeeds; this is a general cryptanalytic approach called *key search*. Human beings being what they are, there is a strong tendency for people to choose relatively short and simple passwords that they can remember. Given free choice, most people will choose their passwords from a restricted character set (e.g. all lower-case letters), and will often choose words or names. This human habit makes the key search job a great deal easier.

The critical factor involved in key search is the amount of time needed to encrypt a potential password and to check the result against an entry in the password file. The running time to encrypt one trial password and check the result turned out to be approximately 1.25 milliseconds on a PDP-11/70 when the encryption algorithm was recoded for maximum speed. It is takes essentially no more time to test the encrypted trial password against all the passwords in an entire password file, or for that matter, against any collection of encrypted passwords, perhaps collected from many installations.

If we want to check all passwords of length n that consist entirely of lower-case letters, the number of such passwords is 26^n. If we suppose that the password consists of printable characters only, then the number of possible passwords is somewhat less than 95^n. (The standard system "character erase" and "line kill" characters are, for example, not prime candidates.) We can immediately estimate the running time of a program that will test every password of a given length with all of its characters chosen from some set of characters. The following table gives estimates of the running time required on a PDP-11/70 to test all possible character strings of length n chosen from various sets of characters: namely, all lower-case letters, all lower-case letters plus digits, all alphanumeric characters, all 95 printable ASCII characters, and finally all 128 ASCII characters.

n	26 lower-case letters	36 lower-case letters and digits	62 alphanumeric characters	95 printable characters	all 128 ASCII characters
1	30 msec.	40 msec.	80 msec.	120 msec.	160 msec.
2	800 msec.	2 sec.	5 sec.	11 sec.	20 sec.
3	22 sec.	58 sec.	5 min.	17 min.	43 min.
4	10 min.	35 min.	5 hrs.	28 hrs.	93 hrs.
5	4 hrs.	21 hrs.	318 hrs.		
6	107 hrs.				

One has to conclude that it is no great matter for someone with access to a PDP-11 to test all lower-case alphabetic strings up to length five and, given access to the machine for, say, several weekends, to test all such strings up to six characters in length. By using such a program against a collection of actual encrypted passwords, a substantial fraction of all the passwords will be found.

Another profitable approach for the bad guy is to use the word list from a dictionary or to use a list of names. For example, a large commercial dictionary contains typicallly about 250,000 words; these words can be checked in about five minutes. Again, a noticeable fraction of any collection of passwords will be found. Improvements and extensions will be (and have been) found by a determined bad guy. Some "good" things to try are:

- The dictionary with the words spelled backwards.
- A list of first names (best obtained from some mailing list). Last names, street names, and city names also work well.
- The above with initial upper-case letters.
- All valid license plate numbers in your state. (This takes about five hours in New Jersey.)
- Room numbers, social security numbers, telephone numbers, and the like.

The authors have conducted experiments to try to determine typical users' habits in the choice of passwords when no constraint is put on their choice. The results were disappointing, except to the bad guy. In a collection of 3,289 passwords gathered from many users over a long period of time:

15 were a single ASCII character;

72 were strings of two ASCII characters;

464 were strings of three ASCII characters;

477 were string of four alphamerics;

706 were five letters, all upper-case or all lower-case;

605 were six letters, all lower-case.

An additional 492 passwords appeared in various available dictionaries, name lists, and the like. A total of 2,831, or 86% of this sample of passwords fell into one of these classes.

There was, of course, considerable overlap between the dictionary results and the character string searches. The dictionary search alone, which required only five minutes to run, produced about one third of the passwords.

Users could be urged (or forced) to use either longer passwords or passwords chosen from a larger character set, or the system could itself choose passwords for the users.

AN ANECDOTE

An entertaining and instructive example is the attempt made at one installation to force users to use less predictable passwords. The users did not choose their own passwords; the system supplied them. The supplied passwords were eight characters long and were taken from the character set consisting of lower-case letters and digits. They were generated by a pseudo-random number generator with only 2^{15} starting values. The time required to search (again on a PDP-11/70) through all character strings of length 8 from a 36-character alphabet is 112 years.

Unfortunately, only 2^{15} of them need be looked at, because that is the number of possible outputs of the random number generator. The bad guy did, in fact, generate and test each of these strings and found every one of the system-generated passwords using a total of only about one minute of machine time.

IMPROVEMENTS TO THE FIRST APPROACH

1. Slower Encryption

Obviously, the first algorithm used was far too fast. The announcement of the DES encryption algorithm [2] by the National Bureau of Standards was timely and fortunate. The DES is, by design, hard to invert, but equally valuable is the fact that it is extremely slow when implemented in software. The DES was implemented and used in the following way: The first eight characters of the user's password are used as a key for the DES; then the algorithm is used to encrypt a constant. Although this constant is zero at the moment, it is easily accessible and can be made installation-dependent. Then the DES algorithm is iterated 25 times and the resulting 64 bits are repacked to become a string of 11 printable characters.

2. Less Predictable Passwords

The password entry program was modified so as to urge the user to use more obscure passwords. If the user enters an alphabetic password (all upper-case or all lower-case) shorter than six characters, or a password from a larger character set shorter than five characters, then the program asks him to enter a longer password. This further reduces the efficacy of key search.

These improvements make it exceedingly difficult to find any individual password. The user is warned of the risks and if he cooperates, he is very safe indeed. On the other hand, he is not prevented from using his spouse's name if he wants to.

3. Salted Passwords

The key search technique is still likely to turn up a few passwords when it is used on a large collection of passwords, and it seemed wise to make this task as difficult as possible. To this end, when a password is first entered, the password program obtains a 12-bit random number (by reading the real-time clock) and appends this to the password typed in by the user. The concatenated string is encrypted and both the 12-bit random quantity (called the *salt*) and the 64-bit result of the encryption are entered into the password file.

When the user later logs in to the system, the 12-bit quantity is extracted from the password file and appended to the typed password. The encrypted result is required, as before, to be the same as the remaining 64 bits in the password file. This modification does not increase the task of finding any individual password, starting from scratch, but now the work of testing a given character string against a large collection of encrypted passwords has been multiplied by 4096 (2^{12}). The reason for this is that there are 4096 encrypted versions of each password and one of them has been picked more or less at random by the system.

With this modification, it is likely that the bad guy can spend days of computer time trying to find a password on a system with hundreds of passwords, and find none at all. More important is the fact that it becomes impractical to prepare an encrypted dictionary in advance. Such an encrypted dictionary could be used to crack new passwords in milliseconds when they appear.

There is a (not inadvertent) side effect of this modification. It becomes nearly impossible to find out whether a person with passwords on two or more systems has used the same password on all of them, unless you already know that.

4. The Threat of the DES Chip

Chips to perform the DES encryption are already commercially available and they are very fast. The use of such a chip speeds up the process of password hunting by three orders of magnitude. To avert this possibility, one of the internal tables of the DES algorithm (in particular, the so-called E-table) is changed in a way that depends on the 12-bit random number. The E-table is inseparably wired into the DES chip, so that the commercial chip cannot be used. Obviously, the bad guy could have his own chip designed and built, but the cost would be unthinkable.

5. A Subtle Point

To login successfully on the UNIX system, it is necessary after dialing in to type a valid user name, and then the correct password for that user name. It is poor design to write the login command in such a way that it tells an interloper when he has typed in a invalid user name. The response to an invalid name should be identical to that for a valid name.

When the slow encryption algorithm was first implemented, the encryption was done only if the user name was valid, because otherwise there was no encrypted password to compare with the supplied password. The result was that the response was delayed by about one-half second if the name was valid, but was immediate if invalid. The bad guy could find out whether a particular user name was valid. The routine was modified to do the encryption in either case.